Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Jussi Knuuttila

A radio coexistence simulator

Master's Thesis
Espoo, May 27, 2015

Supervisor: Professor Heikki Saikkonen
Advisor: Vesa Hirvisalo D.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Jussi Knuuttila

Title:
A radio coexistence simulator

Date: May 27, 2015 **Pages:** 77

Major: Embedded Systems **Code:** T-106

Supervisor: Professor Heikki Saikkonen

Advisor: Vesa Hirvisalo D.Sc. (Tech.)

Simulation is commonly used to study complex systems without having to observe the actual systems, which might be difficult to observe or which might not even exist yet. Simulating in advance is very valuable, as such systems can be extremely expensive and could take a long time to build. One example of such a system is a cellular handset that utilizes new radio technology.

We built a light-weight discrete-event simulator to study the in-device radio coexistence (IDC) problem, which arises when different radios in the same device interfere each other. We studied IDC especially in the context of 3GPP LTE and the 802.11 family of wireless LAN. The use of simulation was motivated by both the need for rapid experimentation with different mitigation strategies and, at the time, the absence of commercial hardware with the required capabilities.

The simulator was successfully used to build a detailed model comprising of a mobile multi-radio device in combination with an LTE base station (eNodeB) and a WLAN access point. The model was then used to develop several coexistence strategies that were measured to improve radio throughput and spectral efficiency due to a decreased amount of failed transmissions due to interference. The developed strategies led to academic publications and two patents.

Keywords: simulation, mobile, radio, coexistence, lte, wlan, discrete event simulation

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Jussi Knuuttila		
Työn nimi:	Radiorinnakkaisuussimulaattori		
Päiväys:	27. toukokuuta 2015	**Sivumäärä:**	77
Pääaine:	Sulautetut järjestelmät	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	TkT Vesa Hirvisalo		

Simulaatiota käytetään yleisesti monimutkaisten järjestelmien tutkintaan ilman, että tarvitsee havainnoida todellisia järjestelmiä, joiden havainnointi saattaa olla haastavaa, tai joita ei voida lainkaan havainnoida koska ne eivät ole vielä edes olemassa. Simulointi etukäteen on hyvin arvokasta, koska monet järjestelmät saattavat olla äärimmäisen kalliita ja pitkäkestoisia rakentaa. Yksi esimerkki tällaisesta järjestelmästä on matkapuhelin, joka käyttää uutta radioteknologiaa.

Kehitimme kevyen diskreetin tapahtumasimulaattorin tutkiaksemme laitteen sisäisen radiorinnakkaisuuden (in-device radio coexistence, IDC) ongelmaa, joka syntyy saman laitteen eri radioiden häiritessä toisiaan. Tutkimme ongelmaa erityisesti 3GPP LTE:n ja 802.11 -perheen langattomien lähiverkkoprotokollien yhteydessä. Simulaation käyttöä tutkimiseen motivoi sekä tarve tehdä nopeita kokeita lievennysstrategioilla että sellaisten kaupallisten laitteiden puute, joissa olisi ollut tarvittavat ominaisuudet.

Simulaattoria käytettiin onnistuneesti yksityiskohtaisen mallin luomiseen, joka käsitti moniradioisen mobiililaitteen yhdistettynä LTE- ja WLAN-tukiasemiin. Mallia käytettiin useiden rinnakkaisuusstrategioiden kehitykseen, joiden mitattiin parantavan radioiden kokonaissuoritustehoa ja spektritehokkuutta pienentyneen siirtovirheiden määrän vuoksi. Kehitetyt strategiat johtivat sekä akateemisiin julkaisuihin että kahteen patenttiin.

Asiasanat:	simulaatio, mobiili, radio, koeksistenssi, lte, wlan, diskreetti simulaatio
Kieli:	Englanti

Acknowledgements

I wish to thank my thesis advisor Vesa Hirvisalo for his seemingly inexhaustible patience, and my girlfriend Lisa Tirkkonen, without whom this thesis would never have gotten completed.

Espoo, May 27, 2015

Jussi Knuuttila

Abbreviations and Acronyms

3GPP	3rd Generation Partnership Project
2G	2nd generation
3G	3rd generation
4G	4th generation
ACK	Acknowledgement
ACP	Adjacent channel power
ADC	Analog-to-digital converter
AM	Amplitude modulation
AP	Access point
API	Application programming interface
ARQ	Adaptive repeat and request
ASK	Amplitude shift keying
AWGN	Additive white Gaussian noise
BB	Baseband
BSR	Buffer status report
CPU	Central processing unit
CSMA/CA	Carrier sense multiple access with collision avoidance
CTS	Clear to send
DAC	Digital-to-analog converter
DCF	Distributed coordination function
DIFS	DCF interframe space
DRX	Discontinuous reception
EIFS	Extended interframe space
FDD	Frequency division duplexing
FM	Frequency modulation
FSK	Frequency shift keying
GLONASS	Globalnaya navigatsionnaya sputnikovaya sistema (Russian)
GNSS	Global navigation satellite system
GPS	Global positioning system

GUI	Graphical user interface
HARQ	Hybrid adaptive repeat and request
HSPA	High speed packet access
IDC	In-device coexistence
IP	Internet protocol
ISM	Industrial, scientific and medical
JIT	Just-in-time
JVM	Java virtual machine
LAN	Local area network
LNA	Low-noise amplifier
LO	Local oscillator
LTE	Long-term evolution
MAC	Media access control
MIMO	Multiple-input and multiple-output
MS	Mobile station
NACK	Negative acknowledgement
NAV	Network allocation vector
OFDM	Orthogonal frequency division multiplexing
OFDMA	Orthogonal frequency division multiple access
OPT	Optimistic partial transaction
PA	Power amplifier
PAPR	Peak-to-average power ratio
PHY	Physical interface
PM	Phase modulation
PS-Poll	Power saving poll
PSK	Phase shift keying
QAM	Quadrature amplitude modulation
QPSK	Quadrature phase shift keying
RAM	Random access memory
RCOEX	Radio coexistence
RF	Radio frequency
RTL	Register-transfer level
RTS	Ready to send
RX	Reception
SC-FDMA	Single-carrier frequency division multiple access
SIFS	Short interframe space
SISO	Single-input and single-output
SNR	Signal-to-noise ratio
SPICE	Simulation program with integrated circuit emphasis
SR	Scheduling request
TCP	Transmission control protocol

TDD	Time division duplexing
TLB	Translation lookaside buffer
TX	Transmission
UE	User equipment
W-CDMA	Wideband code division multiple access
WLAN	Wireless local area network
VoIP	Voice over IP

Contents

Abbreviations and Acronyms **5**

1 Introduction **11**

2 Radio systems **13**
2.1 Radios 13
2.1.1 General concepts 13
2.1.2 Radio hardware 14
2.1.2.1 The transmitter 15
2.1.2.2 The receiver 16
2.1.3 Noise, distortion and interference 16
2.1.3.1 Noise 17
2.1.3.2 Distortion and linearity 17
2.1.3.3 External interference 18
2.1.4 Modulation and error correction 18
2.1.4.1 Basic modulation techniques 19
2.1.4.2 Quadrature amplitude modulation and constellations 19
2.1.4.3 Orthogonal frequency division multiplexing and multiple access 20
2.1.4.4 Error correction 21
2.2 LTE 22
2.2.1 Modulation and coding 23
2.2.2 Frame structure 24
2.2.2.1 Synchronization and timing advance 25
2.2.2.2 Broadcast and random access channels 25
2.2.3 Duplexing 26
2.2.4 Scheduling 27
2.2.5 Hybrid adaptive repeat and request 27
2.2.6 Discontinuous Reception 28
2.3 802.11 29

2.3.1 WLAN architecture . . . 30
2.3.2 Media access control using CSMA/CA . . . 30
2.3.2.1 Carrier sensing . . . 30
2.3.2.2 Backoff and contention . . . 31
2.3.2.3 RTS/CTS and transaction frame spacing . . . 32
2.3.3 Power saving mode . . . 32

3 Radio coexistence 34
3.1 Coexistence strategies . . . 35
3.1.1 Unmanaged coexistence . . . 35
3.1.2 Anticipation and information sharing . . . 35
3.1.3 Traffic shaping . . . 36
3.1.4 Priority override . . . 36
3.1.5 Optimistic partial transactions . . . 37
3.1.6 Scheduler interaction . . . 38
3.1.7 Link adaptation . . . 38
3.1.8 Coexistence strategies with scheduled radio protocols . 39

4 System analysis and simulation 40
4.1 System analysis . . . 40
4.2 Simulation . . . 42
4.3 Simulators . . . 43
4.3.1 Programming model . . . 44
4.3.2 Discrete event simulators . . . 45

5 The RCOEX simulator 47
5.1 Design goals . . . 47
5.2 Implementation overview . . . 48
5.3 Configuration and launcher . . . 48
5.4 Tasks and events . . . 49
5.4.1 Waiting primitives . . . 49
5.4.2 Event firing and reception . . . 50
5.4.3 Reception timeouts . . . 53
5.4.4 Interrupts . . . 54
5.4.5 Implementation . . . 55
5.4.5.1 Tasks and threads . . . 55
5.4.5.2 Simulation cycles . . . 55
5.4.5.3 Primitive implementation . . . 57

6 Simulating in-device radio coexistence **58**
6.1 Coexistence cases . . . 58
6.2 Models . . . 59
6.2.1 Radio models . . . 59
6.2.2 Interference model . . . 59
6.3 Implementation . . . 60
6.3.1 Implementation overview . . . 60
6.3.1.1 Radio models and pipes . . . 60
6.3.1.2 Layer 2 events . . . 61
6.3.2 Abstract protocol and PHY . . . 62
6.3.3 Workload generation . . . 62
6.3.4 Interference caster . . . 63
6.3.5 Tracing and analysis . . . 63

7 Discussion **66**
7.1 Design goals . . . 66
7.1.1 Ease of simulation development . . . 66
7.1.2 Performance . . . 67
7.1.2.1 Benchmarks . . . 68
7.1.2.2 Future optimization . . . 69
7.1.3 Easy reproducibility . . . 70
7.2 Implementation schedule . . . 70
7.3 Simulator features . . . 71

8 Conclusions **72**
8.1 Conclusions . . . 72
8.2 Future work . . . 73

Chapter 1

Introduction

This thesis describes the development and use of a discrete-event simulator and an associated model for studying the in-device radio coexistence problem.

In-device radio coexistence, or IDC for short, is the situation where a device, such as a cellular phone, contains several radio transceivers in close proximity to each other. If these transceivers operate on certain problematic frequency bands relative to each other, it is possible that the radios interfere each other. We call this kind of intereference in-device interference.

Up until recently, in-device interference hasn't been a serious problem, because the 3G radios prevalent in cellular phones, such as W-CDMA and HSPA, have operated at safe frequency bands relative to the industrial, scientific and medical (ISM) radio band widely used for wireless communication protocols such as WLAN and Bluetooth. However, with the advent of 4G and the 3GPP Long-term Evolution (LTE), modern mobile radios sometimes now operate on frequency bands which can interfere and are sometimes interfered by devices which use the ISM band.

When left unmitigated, in-device coexistence can significantly lower the effective throughput of the radios in a device, leading to wasted spectrum and energy. On the other hand, if the radios in the device can be made to co-operate so that they proactively avoid overlapping transmissions, overall throughput can be greatly improved. We call this kind of cooperation a coexistence strategy.

This thesis was made as a part of a project to research these kinds of coexistence strategies. A substantial part of this research involved experimentation to see how different radio protocols interact in an in-device environment. Experimentation with actual hardware would have been highly problematic due to reasons including hardware availability, inability to experiment with cellular base stations and available project resources, so a computer simulator was developed instead.

We chose to simulate coexistence on an abstract protocol level, in which the simulation deals with abstract protocol packets and the lower level details, such as radio symbols and analog signals, are not modeled. This was done both due to the significant difficulty of modeling such concepts and because their accurate modeling is not crucial for studying in-device coexistence. Instead of accurate analog modeling, we used a conservative approximation where all interference fully interferes the affected signals. As real-world interference does not always result in the signal being fully lost, this approximation should be pessimistic. Furthermore, coexistence strategies that can cope with the conservative interference model should also perform well in the real world.

As no suitable and publicly available simulators were available to us, we developed the Radio Coexistence (RCOEX) simulator, a custom process-oriented discrete-event simulator for simulating radio hardware on an abstract protocol level. The simulator is implemented in the Java programming language. It supports programming simulation models directly using Java, as well as higher level configuration of simulation parameters.

The main contribution of this thesis is the design and implementation of the simulator core, consisting of task, event and configuration subsystems, of the RCOEX simulator. This thesis was a part of a successful research project on coexistence strategies for radio communication inside a cellular device. In particular, coexistence strategies were developed for 3GPP Release 8 (LTE) and 802.11 (WLAN) coexistence within a single mobile multi-radio device, and these strategies were measured to improve overall radio throughput and spectral efficiency in simulation. The research led to two academic publications [18, 19] presenting our findings and measurements, and two patents [20, 35].

The structure of this thesis is as follows. In chapter 2 we review the basics of radio transceivers and provide a short overview of the WLAN and LTE protocols. In chapter 3 we present the problem of in-device radio coexistence and introduce some strategies for managing it. In chapter 4 we review the principles of computer simulation in general. In chapter 5 we introduce the main features and some implementation details of our simulator. In chapter 6 we introduce the problem of in-device radio coexistence and how we modeled this using our simulator. Finally, in chapter 7 we discuss our results and in chapter 8 we draw our conclusions and offer suggestions for future work.

Chapter 2

Radio systems

2.1 Radios

As the in-device radio coexistence problem is caused by the physical properties of radios, understanding basic radio operating principles is highly useful for understanding IDC. In this section we provide a basic overview of radios and how they generally operate. For more details, the interested reader can refer to [28], which most of this section is based on.

2.1.1 General concepts

Most radio systems communicate by manipulating high-frequency sinusoidal electromagnetic signals called *carriers* or *carrier signals*. This manipulation, called *modulation*, encodes information into variations of the carrier's phase, frequency and amplitude. [28] lists several good practical reasons for the use of modulated high-frequency sine waves:

- To achieve a good gain, the size of the antenna should be a significant fraction of the signal wavelength. In practice, this means that the frequency should be fairly high.
- Usually, it is legal to transmit only on certain frequencies. These frequencies are controlled and allocated by national or international organizations.
- It is sometimes convenient for the design of the radio hardware.

Even though high-frequency carrier signals are desirable for the actual transmission, such signals are difficult to process in hardware. Consequently, radios usually convert the signals between the low and the high frequencies

and process signals at low frequencies. This low frequency area is called the *baseband* (BB), and the high frequency is called the *radio frequency* (RF). Additionally, the frequency area, also called a *frequency band* might be subdivided into smaller areas called *channels*. We say that frequencies inside and outside the frequency band of interest are *in-band* and *out-of-band*, respectively.

Radio architectures can be classified as being either analog or digital. An analog architecture transmits and receives analog waveforms like sound, while a digital architecture transmits and receives information in the form of bits. Since radio waves are fundamentally analog signals, all radios are at least partially analog devices. However, whereas an analog architecture deals wholly in analog signals, a digital architecture converts between analog and digital by means of analog-to-digital converters (ADC) and digital-to-analog converters (DAC).

Different radio architectures and techniques can be compared in terms of performance, which can be either in terms of operating range, quality, speed, or power efficiency. The operating range is the maximum possible distance the radio is able to satisfactorily operate at. Quality is the ability of the receiving radio to faithfully reproduce the transmitted signal in the face of interference. Speed, for a digital architecture, is the amount of bits that can be transmitted per unit of time. Quality and speed are related in the sense that we can achieve larger speeds if we can improve the quality. Finally, power efficiency is the amount of power consumed in relation to the amount of data transmitted.

As we are interested in radios in the context of in-device radio coexistence in modern mobile devices, we shall mainly focus on digital radio architectures since all radios in these devices are digital.

2.1.2 Radio hardware

The operation and the corresponding structure of a radio can be logically divided into four parts. Radios usually have different signal paths for transmission (TX) and reception (RX), called the transmitter and receiver, respectively. The TX and RX paths can be further divided into parts dealing with the high radio frequencies and baseband frequencies, called the RF section or front end, and baseband section.

The RF section, also called the front end, is the intermediary between the antennae and the baseband section. Because the RF section necessarily deals both with high frequencies and wildly varying power levels (the power difference between a transmitted and a received signal can be as high as 120 dB), it is the most demanding part of the entire radio.

The baseband section modulates information into or demodulates it from a carrier signal. This is called *analog modulation* if the information is an analog signal, and *digital modulation* if the information is a stream of bits. In the case of digital modulation, the baseband section can be further divided into the *analog baseband* and the *digital baseband*, which are separated by D/A converters in the transmitter and A/D converters in the receiver.

Below, we describe the RF and baseband sections of the transmitter and receiver of a simple digital radio architecture. For actual radios, the boundaries of the RF and baseband sections might not always be clear. For example, in some architectures modulation and upconversion (and the corresponding demodulation and downconversion) are performed simultaneously, although we present them below as separate operations residing in separate sections.

2.1.2.1 The transmitter

In the transmitter, the baseband section modulates the signal and converts it into analog form for the RF section, which then upconverts the signal to the carrier frequency and amplifies it to the necessary power level for transmission.

At first, the input signal is in the form of a bit stream of ones and zeroes. It goes through the modulator, which employs a digital modulation scheme to form the modulated carrier signal. Then, a D/A converter converts the signal into an analog waveform to be transmitted by the TX front end.

Then, in the front end, a *mixer* upconverts the signal. A mixer is a component that shifts a signal from one frequency band to another. In the transmitter, it shifts the baseband signal to the carrier frequency band.

The mixer requires a frequency reference to function, which is generated by a *local oscillator* (LO). The LO might be adjustable if the radio is to operate on several carrier frequencies, or it might be fixed.

Upconverting the signal introduces additional noise. In order to avoid amplifying this noise, a bandpass filter discards the noise before amplification.

Actually transmitting the signal over the air requires much more power than the original signal contains. To achieve the necessary power, the signal goes through a *power amplifier* (PA). Because the power amplifier is usually the biggest power consumer in the radio, we generally want it to have good power efficiency. The transmitted signal might, however, restrict the choice of PA, as different kinds of modulation have different linearity requirements. For example, frequency shift keying tolerates nonlinearity much better than phase shift keying.

After the power amplifier, to avoid interfering other frequency bands, a

second bandpass filter filters the signal before it finally passes to the antennae for the actual transmission.

2.1.2.2 The receiver

In the receiver, the RF section listens to the signal from the antennae, and then amplifies and downconverts it for the baseband section, which then converts it into digital form and demodulates it, forming a digital bit stream as a result.

The reception starts from the antennae. They convert electromagnetic waves into a weak voltage signal that is then processed further. However, this signal probably contains out-of-band frequencies. To avoid amplifying these, which would effectively be noise for the actual transmission signal, the signal is first filtered by a bandpass filter that admits only the desired frequency band.

Although the signal now contains only desired frequencies, it is still too weak for processing, and must be amplified before downconversion. To avoid exacerbating noise-related problems later on, the receiver uses a *low-noise amplifier* (LNA).

After amplification, a mixer converts the amplified signal from the carrier frequency to baseband, again using a mixer in combination with a local oscillator. If both the transmitter and the receiver utilize the same carrier frequency (which might not always be the case), it is possible and common to share the local oscillator between paths.

Once the signal is in the baseband, an A/D converter converts the signal into digital form. The demodulator then demodulates and error corrects the signal to yield the final output bit stream.

Although the baseband sections presented here are rather simple at the conceptual level, the modulation and error correction schemes can be rather complex. We defer the study of modulation and error correction to subsection 2.1.4.

2.1.3 Noise, distortion and interference

Radios, like all electronic systems, have to deal with noise and interference. *Noise* is the random variation present in all real-world electronic signals, and distinct from *interference*, which is unwanted variation caused by other signals.

2.1.3.1 Noise

Since all real-world electronic components generate noise, noise is unavoidable. However, circuits can be designed to generate less noise and to be less sensitive to noise.

A very important noise-related number is the *signal-to-noise ratio* (SNR). It specifies how strong our signal is compared to the amount of noise in the system. A low SNR means that it is hard to distinguish our signal from the noise, which might mean that we are unable to correctly extract the information from the signal.

Not all kinds of noise are equal. Different kinds of noise differ by their *spectral density*, which specifies how the noise power is distributed across the spectrum. Some common kinds of noise are *white noise*, which is equally distributed across all frequencies, and $1/f$ *noise*, which is strongest at low frequencies and gets weaker at high frequencies.

A particularily important kind of noise is *additive white Gaussian noise* (AWGN), which is frequently used to characterize background noise of a radio transmission due to factors such as cosmic radiation and thermal noise in the circuits.

Finally, it is common to describe components in radio systems in terms of their *noise figures*. The noise figure measures how much the SNR of a signal degrades as it passes through the component.

2.1.3.2 Distortion and linearity

Ideally, we would like all amplifiers to be perfectly linear. However, all real hardware is nonlinear at least to some degree. While good linearity can be achieved, it is usually possible only for some specific frequency band and at the cost of other desirable features like power efficiency and the amount of amplification.

When a signal passes through a nonlinear amplifier, it is distorted, introducing noise. One kind of distortion resulting from this, which is especially relevant to IDC, is *harmonic distortion*, which introduces frequency components at integer multiples of the original signal. For example, a 1.2 GHz signal will have harmonic components at frequencies 2.4 GHz, 3.6 GHz and so on.

Intermodulation interference introduces frequency components at frequencies that are linear combinations of the frequencies in the original signals. Of particular interest is the *third-order intermodulation interference*, which for given frequencies ω_1 and ω_2 occurs at frequecies $2\omega_1 - \omega_2$ and $2\omega_2 - \omega_1$. If ω_1 and ω_2 are near to each other, these third-order intermodulation components

will also be on nearby frequencies, possibly in the same or neighboring channel. Third-order intermodulation interference is so important that there is a measure for amplifiers, the *third-order intermodulation intercept point* (IP_3), that quantifies how much a given amplifier exhibits this effect.

In addition to applying to amplifiers, real-world filters also exhibit similar effects.

2.1.3.3 External interference

Radios can receive external interference from many sources, including cosmic radiation and atmospheric conditions. Usually, however, the most important interference source for radios is other radios.

Other in-band radios are an obvious source of interference. Some of them that transmit on neighboring channels will often leak some power to the channel of interest. The extent to which this happens is quantified by a measure called *adjacent channel power* (ACP).

In-band interference is especially relevant for radios operating in the *industrial, science and medical* (ISM) frequency band due to the large amount of devices that operate in that band. Devices that utilize the ISM band include not only wireless LANs, Bluetooth devices, and cordless phones, but also non-radio devices like microwave ovens.

In addition to in-band radios, out-of-band radios might also interfere with our signal. Although we try to filter out-of-band frequencies during reception, out-of-band transmitters can also generate frequency components in our band and channel due to harmonic distortion and intermodulation interference. This kind of interference can be very hard to filter.

2.1.4 Modulation and error correction

Modulation is the process of manipulating the properties of a carrier wave to carry information. *Demodulation* is the reverse, extracting information from the properties of a received waveform.

A modulated waveform can be divided into *symbols*. A symbol is some state of the waveform (e.g. a particular amplitude or frequency) that carries a fixed amount of information, measured in bits. Each symbol lasts for a fixed time, and the amount of symbols per unit time is called the *symbol rate*.

In digital radio communication, the modulator encodes information into symbols which are then transmitted, and the demodulator tries to extract the same information based on the received symbols.

2.1.4.1 Basic modulation techniques

A sine wave has three fundamental properties that completely describe it: amplitude, phase and frequency. Correspondingly, there are the three basic modulation techniques, each encoding information into one of these properties. The analog versions are called *amplitude modulation* (AM), *phase modulation* (PM), and *frequency modulation* (FM), while the digital versions are called *amplitude shift keying* (ASK), *phase shift keying* (PSK) and *frequency shift keying* (FSK). We concentrate on the digital versions here. An amplitude shift keying modulator assigns specific amplitude values for ones and zeroes. For each input bit, we transmit the carrier at the corresponding amplitude for one symbol period. To perform the reverse operation, the ASK demodulator measures the average amplitude during each symbol period and compares it to the amplitudes for one and zero. The output will be the bit whose amplitude is closer to the measured amplitude.

A phase shift keying modulator assigns a phase shift for ones and zeroes. Usually opposite phase shifts are assigned, such as $0\,^\circ$ and $180\,^\circ$ or vice versa. For each input bit, we transmit the carrier at a constant amplitude and frequency, but with the corresponding phase shift added. For the common case of $0\,^\circ$ and $180\,^\circ$, this reduces to transmitting either the unmodified carrier or the inverted carrier (i.e., $y(t)$ or $-y(t)$). To perform the reverse operation, the PSK demodulator examines the phase of the received signal compared to a reference. The output will be the bit that better matches the received phase difference. However, for this to work, the reference signal of the demodulator needs to be synchronized to the carrier of the modulator.

A frequency shift keying modulator alternates the carrier between two different frequencies, one of which corresponds to one and the other to zero. The FSK demodulator compares the received signal to two different references, and outputs either one or zero depending on which reference is a better match. A notable advantage of FSK is that since the information is essentially carried in the amount of zero crossings of the transmitted signal, an FSK system can tolerate more nonlinearity in the power amplifier.

2.1.4.2 Quadrature amplitude modulation and constellations

The amount of symbols that can be transmitted using a certain amount of bandwidth is limited by the Nyquist rate [25]. Since our bandwidth is usually limited, there exists a maximum symbol rate that we can hope to achieve. If we want to transmit more bits per second than we can transmit symbols, we have to transmit several bits in each symbol. One common way to do this is *quadrature amplitude modulation* (QAM) [5].

As opposed to phase shift keying techniques, which only transmit one bit per symbol, QAM works by dividing the input bit stream in larger blocks of several bits each, and then transmitting each block in a single symbol. The blocks are usually some power of two in length, and the length is indicated as part of the name of the modulation scheme. Some common QAM schemes are 4-QAM, which is usually called *quadrature phase shift keying* (QPSK), 16-QAM and 64-QAM.

QAM makes use of a concept called I/Q *data*, where the instantaneous state of a sine wave is decomposed into two components, the phase and the amplitude. When these components are interpreted as polar coordinates, the corresponding Cartesian x coordinate is called the *in-phase* data, or I, and the y coordinate is called the *quadrature* data, or Q. I/Q data is often represented by complex numbers such that I corresponds to the real part and Q corresponds to the imaginary part. In this context, the complex plane is called the IQ plane.

The QAM modulator maps each possible input bit pattern to some point in the IQ plane. This arrangement is called a *constellation*. Usually the points are arranged in a rectangular grid, but it is also possible to use, for example, a circular arrangement. When transmitting a bit pattern, the modulator determines the corresponding I and Q coordinates, which are then used to amplitude modulate the in-phase and quadrature carriers. To perform the reverse operation, the QAM demodulator correlates the received signal with the in-phase and quadrature references, giving the I and Q coordinates. Then it outputs the bit pattern whose point in the IQ plane is closest to the received coordinates.

Noise and interference manifest in QAM as offsets in the demodulated I and Q coordinates. The more dense our constellation is, the less distance there is between points, and the larger relative errors a given noise level introduces. Denser constellations therefore have a worse bit error rate but better speed. This makes QAM a straightforward way to turn better transmission quality into more speed.

2.1.4.3 Orthogonal frequency division multiplexing and multiple access

Orthogonal frequency division multiplexing (OFDM) [6, 34] is a common technique to subdivide a frequency band or channel into smaller pieces called subcarriers. As the name suggests, these subcarriers are chosen to be orthogonal, which means they don't interfere with each other. Each subcarrier can then be used in parallel to transmit symbols, modulated using some modulation scheme such as QAM.

Although OFDM necessitates a slower symbol rate, OFDM has many advantages. Because of parallel data streams, it is able to transmit information near the Nyquist rate, making it very spectrum efficient. The slow symbol rate also means that intersymbol interference elimination using guard intervals becomes affordable because guard intervals are short relative to the symbol period.

Because OFDM transmissions are spread across many frequencies, OFDM is also able to tolerate severe frequency-specific interference conditions. If some subcarriers have worse conditions than others, it is possible to use more robust modulation for those subcarriers, while still using high speed modulation, such as 64-QAM, for subcarriers with good conditions. Additionally, OFDM transmissions themselves have a very flat spectrum, and they appear as white noise to other transmissions, a relatively benign form of interference.

OFDM also facilitates the use of a frequency band by many radios simultaneously. It is possible to assign subcarriers exclusively to some radios and other subcarriers to other radios. This enables, for example, a base station to easily communicate with many cellular phones at once. This technique of using OFDM to enable multiple radios to use the frequency band simultaneously is called Orthogonal Frequency Division Multiple Access (OFDMA).

Finally, yet another advantage of OFDM is its suitability for very efficient digital implementation using the Fast Fourier Transform algorithm.

2.1.4.4 Error correction

Noise in a received signal results in the demodulator making wrong decisions and manifests as bit errors, i.e. ones in place of zeroes and vice versa. As radios frequently have to operate in noisy conditions, a coping method is required. One method used by many digital radios is *forward error correction*, where we add redundant error correction bits to every transmission. Even though some transmitted bits are corrupted, we might still be able to recover the correct bits with the help of the error correction bits, thus avoiding a costly retransmission.

Most error correction codes used in radios are either convolutional codes or block codes. *Convolutional codes* are codes that depend only on some amount of previously transmitted symbols. *Block codes* operate on fixed size blocks of information.

Modern radios use Turbo coding [3], where a recursive convolutional code is decoded iteratively using likelihood estimates. Turbo codes are attractive because of their very good performance, making it possible to transmit data near the Shannon limit, combined with the relative simplicity of decoding them in hardware.

2.2 LTE

The 3rd Generation Partnership Project (3GPP) Release 8 is a global standard for telecommunications. It is primarily intended for mobile phones, tablets and similar handheld devices, along with the associated base stations and network infrastructure. Release 8 is more often called by its other name, Long Term Evolution (LTE), and sometimes marketed as "4G" (cf. the use of "3G" for the previous generation of 3GPP standards). We will use the abbreviation LTE to refer to Release 8. In this section, we present the basics of LTE protocol operation, as described in [1, 11, 14], to which we direct the interested reader.

LTE is very focused on efficiency, even at the cost of implementation complexity. Of particular interest is spectral efficiency. Since LTE is usually operated by mobile operators that have paid a large sum of money for their band of radio spectrum, those operators desire good performance out of it. Unsurprisingly, LTE expends significant effort to utilize the radio spectrum very efficiently.

LTE also strives to be scalable in the sense that the available system bandwidth (which depends on the amount of radio spectrum and transmission conditions) can seamlessly transition from offering fast data rates to few users to offering acceptable data rates to a large amount of users, and vice versa. In addition, LTE aims to minimize user latency, something that has been problematic with earlier mobile standards.

Although the many advanced techniques employed by LTE drive implementation complexity up, LTE tries to concentrate this complexity to the base station, which is called the *eNodeB* in LTE terminology, as much as possible. The amount of eNodeBs is relatively small compared to the amount of mobile devices, which are called User Equipment (UE), and it will typically be connected to the main power grid, which means that the eNodeB is not constrained by cost or by power to the same degree as the UE is.

A significant difference between LTE and its predecessors is that LTE is purely packet switched, which means that all traffic is sent in discrete, independent packets. This is in contrast to circuit switching, where a dedicated connection can be set up between two endpoints to guarantee some data rate and latency. Earlier 3GPP releases used a combination of packet and circuit switching, the latter typically being used for voice communication. In LTE, voice communications are also packet switched, and the system relies on the eNodeB scheduler to provide good enough quality of service (QoS).

2.2.1 Modulation and coding

LTE employs different modulation methods in its downlink (eNodeB to UE) and uplink (UE to eNodeB). These methods have different tradeoffs, and using both instead of just one allows the tradeoffs to be made so that UE complexity and power concerns are minimized.

In the downlink direction, LTE utilizes Orthogonal Frequency Division Multiple Access (OFDMA). OFDMA along with the closely related OFDM were both briefly introduced in subsection 2.1.4.3. OFDMA is very attractive for LTE downlink use, as it combines very high spectral efficiency with simple receiver design. The use of OFDMA also makes it possible to allocate non-contiguous frequency bands (i.e. resource blocks, which will be introduced later in this section) for a single UE, granting more freedom to the eNodeB scheduler, and thus also improving total system performance on average.

In LTE, OFDMA is used with symbols that are narrow in the frequency domain and relatively long in the time domain. A separate cyclic prefix is used for each symbol, obviating the need for handling intersymbol interference in the UE receiver. On the transmitter side, the OFDMA signal is transmitted using multiple physical carrier frequencies, which results in the waveform having a high peak-to-average power ratio (PAPR). This, in turn, requires good linearity from the power amplifier, which means bad power efficiency. However, in the case of LTE, the transmitter power amplifier is in the eNodeB, where amplifier cost and power usage are more acceptable.

In the uplink direction, LTE uses Single-Carrier Frequency Division Multiple Access (SC-FDMA). [23, 24, 27] SC-FDMA is closely related to OFDMA conceptually: in both techniques, a large frequency band is split into smaller orthogonal sub-carriers. Where SC-FDMA differs, however, is that each transmitter transmits using only a single physical carrier frequency, and the sub-carriers are purely virtual (i.e. they exist only in the inverse Fourier transform inputs used to generate the time domain signal, and not as physical carrier waves). Like in OFDMA, a transmitter may utilize one or more sub-carriers, but in SC-FDMA they must be adjacent, forming a contiguous frequency band. Each transmitter, then, has a contiguous frequency band which it uses to transmit symbols which are potentially wide in the frequency domain but relatively short in the time domain. Although single transmitters are restricted to contiguous bands, multiple transmitters (i.e. multiple UEs) can still transmit simultaneously on separate bands, which can be received normally by the receiver (i.e. the eNodeB) similarly to OFDMA.

The advantage of SC-FDMA lies in the fact that transmitting the signal using just a single physical carrier results in the signal having a good PAPR, and thus imposing lighter restrictions on the quality of the trans-

mitter power amplifier. This means that the PA in the UE can have good power efficiency, which is important for battery-powered devices. However, because SC-FDMA uses short symbols, the use of a separate cyclic prefix for each symbol (as in the downlink direction) would have an unacceptably high overhead. Therefore, SC-FDMA uses cyclic prefixes only between blocks of several symbols. Unfortunately, this means that the blocks suffer internally from intersymbol interference, which must be addressed using an equalizer in the receiver, driving receiver complexity up. Again, in this case, the receiver is in the eNodeB, where additional complexity is more acceptable.

Fom encoding the data stream in both downlink and uplink directions, LTE uses 1/3 rate Turbo coding [3] with block sizes of up to 6144 bits. Turbo coding was briefly introduced in subsection 2.1.4.4. Although Turbo decoding has very significant processing requirements, the high performance of Turbo codes is very helpful in achieving LTE's spectral efficiency goals, in addition to being well suited for HARQ soft recombining as described later in this section.

2.2.2 Frame structure

The eNodeB scheduler is responsible for allocating all radio resources for UE use, except the random access channel. The unit of these allocations is the *resource block*. A resource block is 180kHz (equal to 12 subcarriers) in the frequency domain and 1ms in the time domain. The reason why resource blocks contain 12 subcarriers instead of just one is that the signaling overhead for transmitting scheduling information would otherwise be needlessly large.

Resource blocks form larger units called *frames*. Each frame is 10ms long, consisting of 10 *subframes*, each of which is 1ms long. Each subframe is a single resource block in the time domain, and a varying amount of resource blocks in the frequency domain. Then, the total amount of resource blocks in a frame depends on the total width of the frequency band or bands in use. Some of these resource blocks are taken by reference signals for channel estimation, and some contain eNodeB signaling information, but most are allocated for UE use by the eNodeB scheduler.

Each resource block can be modulated differently, which allows the system to exploit differing transmission characteristics. Interference might be dependent on frequency (e.g. high-power transmitter on a nearby frequency band) or time (e.g. strong temporary interference). Additionally, UEs in different locations might experience different interference characteristics. For example, one UE might be near a strong short-range narrowband interferer, which makes using resource blocks in that band very undesirable, whereas another UE in a different location might be completely unaffected by that

interferer.

The eNodeB requires UEs to perform channel estimation by sending and receiving pre-determined reference signals, which can then be measured at the other end to estimate the amount of interference present for those UEs at different frequencies. These estimates are used to try to determine which resource blocks are good for which UEs, and to be able to select the best possible modulation scheme for those resource blocks. For UEs with very low interference, a high throughput modulation scheme such as QAM-64 can be used, while UEs that are being heavily interfered might have to resort to QPSK.

2.2.2.1 Synchronization and timing advance

When transmitted, radio waves do not reach the receiver instantaneously. Instead, they travel at the large but finite speed of light. The speed of light in air is approximately 300000km/s, or 300km/ms. As an eNodeBs can potentially service an area that covers tens of kilometers, the signal propagation delay between the eNodeB and various UEs at different distances can vary significantly relative to the time scales the eNodeB scheduling operates at.

In LTE, the eNodeB's clock is authoritative, which means that all scheduling is in terms of the eNodeB's clock, and all UEs are required to synchronize to it. The mechanism by which this is done in LTE is called the *timing advance*.

Timing advance works by the eNodeB measuring the approximate transmit time t_{TA} to each UE. When the UE transmits data that is scheduled to arrive at the eNodeB at time t, it will actually transmit at time $t - t_{TA}$ so the eNodeB receives the transmission at time t. Correspondingly, when the UE receives data that the eNodeB sent at time t, it will attempt to receive it at $t + t_{TA}$.

The measured timing advance is not exact, but it reduces the timing error and the interference caused by neighboring transmissions overlapping in the time domain. Additionally, the timing advance is not constant, as the UEs are often moving physically. To take this into account, the eNodeB can readjust the timing advance if it determines that the current timing advance is not satisfactory.

2.2.2.2 Broadcast and random access channels

When a UE wishes to associate with an eNodeB, the first thing it must do is listen to that eNodeB's broadcast channel if it is known, or scan for broadcast channels if not. The eNodeB periodically transmits system-wide parameters

on the broadcast channel, such as the random access channel parameters for that eNodeB, and synchronization information.

After obtaining the necessary information from the broadcast channel, the UE will next attempt to transmit on the random access channel according to the parameters received. The random access channel is the only radio resource in LTE which is not synchronized or scheduled by the eNodeB. Its only purpose is to allow UEs that don't have any allocated traffic or that have not yet been recognized by the eNodeB to issue a *scheduling request* (SR), which will be further discussed in subsection 2.2.4 below.

2.2.3 Duplexing

In LTE, two-way communication, also called *duplexing*, works differently depending on which frequency band is used. On some bands, *frequency division duplexing* (FDD) is used, while on others, *time division duplexing* (TDD) is used. We say that the LTE radio is in FDD or TDD mode to indicate which is being used.

In FDD mode, the frequency band is divided into two parts, with a guard band in between. One part is for RX and the other for TX. FDD allows truly simultaneous two-way communication, which can improve latencies, while also being simple to implement in the UE. However, FDD can't utilize the entire frequency band effectively if traffic is mostly one-directional, and the guard band wastes a part of the used frequency band. Additionally, FDD requires a higher quality filter in the UE to better separate the RX and TX bands.

In TDD mode, the entire frequency band is used for both uplink and downlink traffic, but each UE alternates between transmitting and receiving. In each LTE frame, there is a split between downlink and uplink subframes that the eNodeB can schedule traffic into. There are a number of TDD configurations with different relative amounts of downlink and uplink subframes to exploit different kinds of traffic patterns. Notably, mobile devices typically receive much more data than they transmit, so it can often be desirable to have, for example, a $9:1$ or $8:2$ split between downlink and uplink subframes. If the eNodeB detects that the average traffic patterns have changed and the current configuration is no longer optimal, the configuration can be changed on the fly. TDD can offer better total spectral efficiency in the face of varying user traffic patterns, at the cost of worse average latency.

2.2.4 Scheduling

When scheduling, the eNodeB can consider many things when making decisions. These can include the amount of data a UE has to send (reported via BSRs, see below), the amount of time a UE has spent waiting to send, the amount of downlink traffic buffered for a UE and the time it has been waiting in the buffer, the estimated channel quality of a UE for different resource blocks, the type of subscription (e.g. user pays extra for higher throughput and/or better latency), the total amount of users, and the estimated total throughput of the system.

After scheduling decisions have been made for a frame, the scheduling information is transmitted to UEs in the control region of that frame, which takes up a varying amount of symbols at the beginning of each frame. To minimize latency and signaling overhead, downlink traffic for a UE can appear in any frame without prior notification, except when DRX (see below) is in use.

UEs must receive and decode the control region of all frames to determine whether there is any downlink traffic for that UE in the rest of the frame. If there is not, the UE can skip receiving and decoding the rest of the frame to conserve energy. If there is traffic, the control region indicates the resource blocks that the UE must decode.

Uplink traffic allocation must be explicitly requested using scheduling requests (SRs). When a UE has traffic to send, it uses the random access channel to send a scheduling request to the eNodeB. The eNodeB then responds with a *grant* which gives the UE permission to transmit some resource block. Since a single resource block can't contain much data, and since requesting each block explicitly would result in huge overhead, the UE attaches a *buffer status report* (BSR) to each transmission. The BSR contains the amount of bytes the UE still wants to send, which the eNodeB uses to issue more grants until the UE sends a BSR of zero. After sending a zero BSR, the UE must send another scheduling request if it wants to send again. The use of grants allows the eNodeB to dynamically dedicate bandwidth to those UEs that need it most, and the use of SRs and BSRs allows large transmissions to be made with minimal overhead.

2.2.5 Hybrid adaptive repeat and request

LTE handles low-level data transmission using *hybrid adaptive repeat and request* (HARQ) processes. All downlink and uplink data is split into packets, each of which is assigned to a single HARQ process. A *HARQ process* can be thought of as an attempt to transmit a single packet, including the associated

retransmissions and acknowledgements. There are 8 HARQ processes, which operate in parallel, each attempting to transmit a separate packet.

Acknowledgements are the fundamental mechanism of HARQ processes. For each transmission, the recipient sends either a positive acknowledgement (ACK) if decoding was successful or a negative acknowledgement (NACK) if decoding failed. The sender, upon receiving a NACK or if no acknowledgement is received, retransmits. When the sender receives an ACK, the HARQ process is completed and a new one begins. There is also a limit for the maximum number of retransmissions. If the HARQ process hasn't completed when the limit is reached, the HARQ process fails and the data is dropped.

The HARQ processes in LTE are based on the principles of soft recombining, chase combining and incremental redundancy [10]. *Soft recombining* is the practice of storing a received transmission even though its decoding fails. Upon retransmission, the new transmission is combined with the previously stored one, resulting in more total information and an increased decoding probability compared to simple retransmissions. When *chase combining* is used, the retransmissions are identical, which results in noise averaging out. When *incremental redundancy* is used, the retransmissions contain additional error correction bits for the original transmission, which enable the receiver to decode more effectively. LTE supports both chase combining and incremental redundancy. The selected soft recombining method can depend on the operating conditions and the used modulation scheme.

HARQ can be used in several modes. The one used in LTE is called the *stop-and-wait* mode, where a HARQ process halts as soon as decoding fails and requests for retransmissions. To hide the resulting latency, LTE uses several parallel HARQ processes, so other processes can complete while some are retransmitting.

The use of HARQ improves LTE reliability and performance. However, it is not foolproof due to the maximum limit for retransmission. This is by design, as some data might expire and not be worth retransmitting, and we might want to transmit newer data instead. If high reliability is required, it can be provided by means of a higher level protocol such as TCP.

2.2.6 Discontinuous Reception

LTE also has a *discontinuous reception* (DRX) mode [4] for reducing power consumption. A UE can request that the eNodeB puts it in DRX mode. In DRX mode, the UE has a configured constant-duration DRX cycle, which is further divided into the *on-time* and the *DRX opportunity*.

During the on-time periods the UE functions as normal, but during the

DRX opportunity periods, the UE is allowed to power down its receiver as soon as it has no more already scheduled traffic and has completed all ongoing HARQ processes. While it is inactive due to DRX, the UE does not need to receive control regions of incoming frames, as it would normally be required to do. The eNodeB knows when the UE is inactive due to DRX, and does not schedule downlink traffic for it during those periods.

Usually, the on-time is configured to be short relative to the DRX cycle, which can allow the UE to be inactive most of the time, which can yield power savings as large as 95%. As a trade-off, DRX increases average latency, as transmissions during inactive periods have to wait until the next on-time period.

2.3 802.11

The 802.11 family of wireless communication standards, often referred to as either Wi-Fi or Wireless LAN (WLAN), are very common means of wireless communication. The 802.11 family of protocols are relatively simple and cheap to implement, and can be supported by many kinds of devices ranging from devices such as music players and cellular phones to laptop computers and game consoles.

There are many standardized versions of the 802.11 protocol in existence. The original standard was published in 1997, and was followed by 802.11a and 802.11b in 1999, 802.11g in 2003 and 802.11n in 2009. At the time of writing, the next version of the protocol, 802.11ac, is being standardized. The later protocols have introduced improved modulation schemes and 802.11n also added MIMO support to significantly improve throughput. From now on, we will refer to the 802.11 protocols collectively as WLAN.

WLAN is designed to be relatively simple to implement, with few mandatory transmissions, which allows it to be supported by cheap devices. The downside to this is that WLAN's spectral efficiency is not as good as some other protocols, such as LTE. However, since WLAN devices mostly operate on the industrial, scientific and medical (ISM) band, which is free to use, spectral efficiency is not as crucial as for devices which operate on dedicated, paid-for radio bands.

A distinguishing characteristic of WLAN is that it uses continuous time. While many other protocols, such as LTE and Bluetooth, divide time into fixed size synchronized pieces and allocate transmissions in those pieces, WLAN transmissions can start at any time and are timed using specified inter-transmission delays.

This section presents the basics of WLAN operation. Specifically, we

present the WLAN network architectures, the WLAN MAC protocol, and the WLAN power saving mode. Most material in this section is based on [12], to which we also direct the interested reader.

2.3.1 WLAN architecture

WLAN networks can be broadly classified into two categories: *infrastructure* networks and *ad hoc* networks.

In infrastructure networks WLAN devices, often called Mobile Stations (MS), communicate with outside networks through a special device called the Access Point (AP). Often the Access Point is a dedicated device, for example a WLAN router, but it can also be something else, such as a cellular phone sharing its LTE Internet connection using WLAN (a process sometimes called *tethering*).

Ad hoc networks, in contrast, are WLAN networks without an Access Point and which operate in a peer-to-peer fashion. Ad hoc networks are relatively uncommon, and we will not consider them further in this section.

2.3.2 Media access control using CSMA/CA

At the Media Access Control (MAC) layer, WLAN uses the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol to control access to the radio medium. In this subsection we describe its operation at a general level, as presented in [8] and [16].

Because wireless devices cannot, in general, detect collisions when transmitting, WLAN devices try to proactively avoid collisions instead of detecting them as they happen, as is done in the Ethernet protocol, for example. The mechanisms used for this are *carrier sensing* (both physical and virtual), *interframe spacing* and *binary exponential backoff*.

2.3.2.1 Carrier sensing

CSMA/CA prohibits devices from starting transmissions if another device is already transmitting, since that would likely cause a collision and failure of both transmissions. When a device wants to transmit, it will first carrier sense to see if another device is transmitting. WLAN devices utilize both physical and virtual carrier sensing.

Physical carrier sensing is done by listening to the radio medium and checking if there is a signal present. Virtual carrier sensing is done using the Network Allocation Vector (NAV), which is a counter present in all WLAN devices. Whenever WLAN devices send frames, they set a special duration

field in the frame to contain the estimated duration of the *transaction* (i.e. the entire sequence of frames and their responses, not just the single frame) in progress. Whenever WLAN devices receive frames not meant for themselves, they set their NAVs to match the duration field. Whenever the NAV is non-zero, the transaction of some other device is highly likely to still be in progress and trying to transmit would probably cause collisions. Thus, whenever the NAV is non-zero, virtual carrier sensing reports the medium as being busy.

2.3.2.2 Backoff and contention

WLAN frames are separated by interframe spaces, which are simply time delays of specified lengths. The three most important of these are, listed from shortest to longest, the Short Interframe Space (SIFS), the DCF Interframe Space (DIFS), and the Extended Interframe Space (EIFS).

Before starting a transmission, a WLAN device is required to first carrier sense for the duration of a DIFS (or, if errors were detected in the previously received frame, the duration of an EIFS instead). If no transmissions are detected during that time, the device is then free to immediately start transmitting. If a transmission is detected, the device will start *contention* using the binary exponential backoff procedure.

If devices were to start transmitting immediately after a transaction is over, each one would find the DIFS period after the transaction empty, and then start transmitting simultaneously, causing a collision. To avoid this, after a transaction ends, each WLAN device that wishes to transmit must select a time period randomly (the *backoff time*) and carrier sense for that period in addition to the DIFS. If several devices are unlikely to choose the same time period, collisions are unlikely.

The random backoff time is chosen uniformly from a range called the *contention window*. The contention window is initially small, so devices will choose small backoff times. Whenever a collision occurs (which is indicated by the recipient failing to acknowledge), the contention window is doubled and a new random backoff time is selected. Each doubling of the contention window makes it exponentially more unlikely for all of the contending WLAN devices to select the same backoff period, so it is very probable that one of them is able to successfully start its own transaction.

Once set, the backoff time is never reset during the backoff procedure unless the device participates in a collision. The backoff time is kept in a counter which is decreased whenever carrier sensing detects that the medium is unused. A device which chose a large backoff time initially will keep decreasing its backoff time until it reaches zero and the device attempts to transmit. This behavior ensures that devices which have already waited for

a long time are more likely to win contention and start transmitting next.

2.3.2.3 RTS/CTS and transaction frame spacing

A WLAN device starts each transaction with a short Ready-to-Send (RTS) frame, which also serves to set the NAVs of all devices that can hear the sender. The recipient replies to this with a Clear-to-Send frame (CTS) which both acknowledges to the sender that no collision occurred, and also sets the NAVs of all devices that can hear the recipient. In the case of the so-called hidden terminal problem, the set of devices that can hear the sender and the recipient can be different, so both the RTS and CTS are required to try to ensure that nearby devices will not try to transmit simultaneously.

After the RTS and the CTS, the sender sends a number of data frames, and the recipient acknowledges each of them with an ACK frame. All frames in a transaction are separated by SIFS time periods. Because a SIFS is shorter than a DIFS, it is guaranteed that the next transaction frame will start before any other potential transmitter has had time to carrier sense for the required DIFS period, essentially giving higher priority to the transaction frames.

2.3.3 Power saving mode

WLAN devices have two operating modes: *infrastructure mode* and *power saving mode*. In infrastructure mode, the device's radio is always powered up and it is always either receiving or transmitting, allowing for maximally fast operation and minimal latency. In power saving mode, the device sleeps by keeping its radio powered off most of the time, powering it up only to transmit and to receive anticipated transmissions. Since radios are significant consumers of power in battery-operated devices, keeping them powered off in this manner can conserve a considerable amount of power and prolong battery life.

Because the radio is powered off most of the time, devices in power saving mode are unable to receive frames from the Access Point unless they expect them. Devices entering power saving mode inform the Access Point, so the Access Point knows to put data for that device in a queue. Devices in power saving mode periodically wake up at predetermined times to listen for beacon frames sent by the Access Point at regular intervals. If the beacon frame indicates that there is queued data for the device, the device next sends a PS-Poll frame to request for it. After receiving a PS-Poll frame, the Access Point responds with the actual data frame. Each frame sent by the Access Point contains information whether there is still more data available, so the

device knows to keep receiving. After there is no more data, the device can immediately resume sleeping until the next known beacon frame.

Chapter 3

Radio coexistence

We define *in-device radio coexistence* or in-device coexistence [15, 17] as the situation where a mobile device contains several radios that need to operate at the same time. Depending on the particular frequencies and radio protocols, these radios may interfere each other. We call this interference caused by other radios in the same device *in-device interference*.

In in-device interference cases, the interfering radio is in the same device and thus very close to the interfered radio. The close proximity means that signals are not attenuated because of distance and interfere at full transmitter power. Since the difference between the transmit power of the interferer and the received signal power of the interfered radio can be as high as 120 dB, the interference can be very strong even if only a small part of the transmitted power leaks to the receiver of the other radio.

Radio *transactions* are exchanges of one or more separate transmissions between radios that form a larger whole. Let's look at WLAN as an example, where a WLAN mobile station polls the access point for data, which then responds by sending the data. After receiving the data, the mobile station sends an acknowledgement. These three separate transmissions form a single coherent transaction. Transactions are a useful concept when discussing in-device interference, because in-device interference often interferes strongly during very specific time periods. This might cause specific transmissions of a transaction to fail, which can lead to the transaction failing as a whole, despite most of its transmissions succeeding.

We also differentiate symmetric and asymmetric in-device interference. The interference between two radios is *symmetric* if both radios interfere each other and *asymmetric* if one of the radios interferes the other but not vice versa.

A prominent cause of problems is nearby frequencies that leak power into adjacent frequencies because of nonidealities in filters and amplifiers such as

intermodulation interference. Problems arising from nearby frequencies are typically symmetric, so coexistence measures have to be implemented in both radios.

One specific symmetric case is the LTE TDD band 40 (2.3 GHz – 2.4 GHz) with the ISM band (2.4 GHz – 2.5 GHz). These bands are right next to each other, so transmitters on either band are likely to interfere with receivers on the other. An example of asymmetric interference would be the LTE FDD band 7, whose TX band is 2.5 GHz – 2.57 GHz, close enough to interfere receivers on the ISM band. The RX band for FDD band 7, however, is 2.62 GHz – 2.69 GHz, which is far enough to not get interfered by transmitters on the ISM band.

A second, more theoretical, cause of problems is frequency bands whose harmonic distortions lie on other important bands, causing difficulties for receivers on those bands. As harmonic distortions always occur on higher frequencies, this type of problem is asymmetric and only the higher frequency band is interfered. Fortunately, the bands in use for mobile radios are for the most part selected so that the harmonics do not occur on important bands.

3.1 Coexistence strategies

3.1.1 Unmanaged coexistence

The simplest possible form of coexistence is the *unmanaged coexistence*, where the radios do not cooperate in any way. It is mostly useful as a baseline for comparison of different coexistence strategies. In practice, we expect that real devices will at least use overriding (see below) and not resort to unmanaged coexistence.

3.1.2 Anticipation and information sharing

Usually, it is possible for a radio to anticipate some or all of its activities at least some time in advance. This information is very useful for making good coexistence decisions. Since coexistence problems concern all radios in a system, it is necessary to share this information between radios.

The most obvious form of anticipation is whether we have outgoing data of our own: if we do not, we will not transmit until we do. However, some protocols might require us to transmit acknowledgements to incoming transmissions that cannot be anticipated. Therefore, while the absence of outgoing data does not absolutely guarantee that we will not transmit, it is useful information nevertheless.

Even though it is impossible to predict sporadic incoming transmissions, we can predict constant known traffic. Examples include things like WLAN beacons and voice calls, where we know that after receiving a piece of data, we do not need to receive more until some period of time has passed.

Regardless of whether we can predict when transactions will start, since the contents of transactions are rigorously specified, we can accurately predict the traffic patterns of those transactions once they have started. Because the transactions generally have a short duration, we can only predict in the short term, on the order of some milliseconds in advance, but this is often enough for other radios to make decisions.

3.1.3 Traffic shaping

When we have data to transmit, but can decide the exact moment the transmission will occur and perhaps the amount of data to transmit, we can utilize traffic shaping. *Traffic shaping* in the context of coexistence is when we delay and prioritize transmissions so they occur at a time where interference is minimized. In practice, this could mean a number of things:

- If we know some other radio is receiving and our transmission is non-critical, we wait until the other radio is done before transmitting.
- If we have choice in what data packet to transmit, e.g. a short packet or a long packet, we can prioritize the shorter over the longer if we know that transmitting the shorter will not interfere with other radios but transmitting the longer will.
- If we know that that transmitting would start a longer transaction of multiple transmissions (e.g. a poll packet, the response, the acknowledgement to the response) and that we cannot receive some part of that transaction because some other radio is transmitting, we don't start that transaction in the first place.

Not all radio protocols are amenable to traffic shaping. In LTE, for example, all transmissions are scheduled by the base station, so little flexibility is left to the UE and most of the above techniques cannot be used. However, the widely used WLAN protocol is very flexible and can utilize all of the above.

3.1.4 Priority override

When we have a critical transmission, but it would either interfere or get interfered by some other less important transmission, we can *override* the

other transmission, interference notwithstanding. If we need to transmit, we transmit right away even if it interferes the other radio. If we need to receive, we shut down the power amplifier of the other radio to stop it from interfering us.

Overriding other radios can be rather disruptive, but it is frequently necessary in, for example, cellular phones, where the LTE radio utilizes expensive spectrum and resource blocks scheduled for us would be wasted if we cannot use them. On the other hand, WLAN utilizes the free-to-use ISM band, so it is very desirable for LTE traffic to override WLAN traffic.

3.1.5 Optimistic partial transactions

When we only have a short interference-free window that is just long enough for our data payload, but not for the following acknowledgement, we can sometimes use optimistic partial transactions. *Optimistic partial transactions* (OPT) are transactions where we know in advance that the acknowledgement to our transmission is very likely to be interfered, and treat the lack of acknowledgement the same as "positive acknowledgement" instead of "negative acknowledgement" as is commonly done.

This can improve performance of radios with coexistence problems because, in good operating conditions, the data payload is likely to be successfully transmitted and the acknowledgement is likely to be positive. If the acknowledgement is in fact negative, a higher protocol layer, such as TCP, will resend the data anyway if necessary, so correct operation is not jeopardized. Essentially, optimistic partial transactions optimize the common case at cost to the exceptional case.

We can only use OPTs if we are the transmitter, and in that case we can always do so, because the receiver side cannot distinguish them from demodulation errors with the acknowledgement. Whether we *should* is another question, but for throughput-oriented data, as opposed to latency-sensitive data, and in good operating conditions, using OPTs can improve overall throughput.

If we are the receiver, we are completely agnostic regarding OPTs. We will always transmit the acknowledgement anyway, unless we are overridden. Whether the transmitter will then interpret the missing acknowledgement as positive or negative acknowledgement is beyond our control. However, radios could conceivably negotiate whether OPTs should be used, although no current radios do.

3.1.6 Scheduler interaction

With some protocols, especially LTE, the mobile device has very little or no control as to when transmissions occur, and must obey whatever instructions are given by the scheduler. In these situations, we would like to be able to inform the scheduler about our in-device coexistence problems, so the scheduler could make decisions that improve our situation without degrading the performance of the system as a whole. In particular, we would like to discourage the scheduler from scheduling certain kinds of traffic at certain times and to disregard in-device interference when choosing modulation schemes.

Normally, the scheduler can allocate to mobile devices whatever resources it deems best for the performance of the whole radio system, while trying to guarantee some amount of minimum performance to each device. However, it might be beneficial for us to accept some performance loss as a tradeoff for significant improvement in the performance of some other radio. We want to inform the scheduler that, if possible, we would prefer to not transmit at certain times, so we can allow other radios to transmit during those times. We call these kinds of time periods *guaranteed unscheduled periods*.

In most cases, the scheduler should be able to honor these requests and allocate the resources in question to some other user. However, a notable exception is the case when a large proportion of users request to avoid the same resource slots. This occurs if the in-device interference follows the same temporal pattern for all users. For example, all of these users might be watching the same live video stream via WLAN, which results in similar traffic patterns and similar avoidance preferences. In this case, to avoid leaving expensive resources unused, the scheduler should deny a proportion of avoidance requests to make sure the resource usage and the resulting overall system performance stays high.

3.1.7 Link adaptation

To optimize throughput, we aspire to use the fastest possible modulation scheme that works well in the current operation conditions. In good conditions, we can use very high speed modulation, such as 64-QAM, but in noisy conditions we cannot demodulate it reliably, so we have to use a slower scheme, such as QPSK. This is called *link adaptation*.

When choosing the modulation scheme, schedulers estimate the operating conditions by counting how often transmissions fail. Transmissions lost due to in-device interference are also taken into account, because the scheduler does not know the reason why the transmission failed. Since the transmissions failed because of in-device interference, they would have failed

regardless of the speed. On the contrary, by reducing the speed we only exacerbate the problem, because the transmissions take longer, which only increases the probability of in-device interference causing more failures. This leads to even more failed transmissions, which causes link adaptation to eventually drop the speed to the minimum possible rate, leading to needlessly bad performance.

This kind of vicious circle is especially bad when it wastes dedicated resources, such as LTE allocations, which could have been used by someone else, thus degrading the performance of the entire system. For this reason among others, LTE will always override in-device interferers: wasting expensive spectrum is inexcusable.

For good performance in in-device interference conditions, we want to make the scheduler disregard failures due to in-device interference or at least adapt the link less aggressively. In some scenarios we might even want to use faster modulation than what would otherwise be optimal, if the gain from being able to utilize time windows that would otherwise be unusable is higher than the loss from the increase in inherent failure rate.

3.1.8 Coexistence strategies with scheduled radio protocols

To be able to handle in-device coexistence scenarios well with radio protocols that have an authoritative scheduler, we need to have means of informing the scheduler of our coexistence problems. The granularity of this information determines how well the scenarios can be handled. The more information the scheduler has, the better decisions it can make.

Unfortunately, although current radio protocols have some coexistence signaling features, they are rarely used in practice. Despite this, features designed for power saving could be used to obtain guaranteed unscheduled periods. Still, issues with link adaptation cannot be handled using power saving features, and remain problematic in low-priority radios that cannot use overriding to avoid it.

Chapter 4

System analysis and simulation

In this chapter, we provide a brief introduction to system analysis, simulation, simulators and their implementation. Simulation is a widely used technique and well studied in the literature. The material in this chapter is largely based on [21, 31], to which we direct the interested reader for more detailed takes on the subject.

4.1 System analysis

The two most fundamental concepts in system analysis are the system itself and the system model. A *system* is defined as the actual situation or process of interest along with all participating entities. A *model* is an abstract representation of the system, often a simplified one.

Throughout this subsection, we look at the *n-body problem* as an example system. In the n-body problem, the system consists of a number of celestial objects such as stars and planets that attract each other gravitationally and move according to Newton's laws.

Systems can be broadly classified into two categories: open and closed. A *closed* system is one that exists in a conceptual vacuum and doesn't interact in any way with external entities. An *open* system is one that exists as a part of a larger environment that it interacts with. The n-body problem can be treated as a closed system if we look at the bodies in a vacuum, but it can also be treated as an open system if we look at the bodies as a part of a larger system. In the open system example, the bodies of interest (the system in question) might constitute a solar system as a part of a galaxy (the environment).

In order to analyze the system, we must construct a mathematical model of the system. As most real-world systems are much too complex to analyze

in their entirety, the model is usually an abstraction. The level of abstraction necessarily depends on the use of the model. We want to abstract everything that is too detailed for the analysis, but not so much that we lose the information we are interested in. In the case of the n-body problem, it is often acceptable to model the bodies as points (as opposed to spheres or more complex shapes), but going further, such as by abstracting also the trajectories, can result in the model being unusable for analyzing the phenomena we are interested in.

Like systems, models can be categorized according to their characteristics. One important categorization is determinism: a model is *deterministic* if it does not depend on any probabilistic inputs, and therefore always produces the same output for a given input. If a model does have probabilistic inputs, it is called a *non-deterministic* model. We note that the determinism of the model does not necessarily correspond to determinism of the analyzed phenomenon. Non-deterministic phenomena can sometimes be modelled deterministically and vice versa.

The most straightforward way to analyze a system is to actually observe it in operation and measure interesting quantities. However, this requires that the system already exists and it is possible to perform the required measurements within the budget and allotted timeframe. Furthermore, it might be necessary to experiment with the system, such as by tweaking some system parameters. All these limitations make measurement of real systems usually an infeasible option. When using this approach with the n-body problem, we are obviously unable to meaningfully alter the system and are thus restricted to just observing the actual motions of the planets and stars, e.g. by telescope.

If the model is simple enough, it might be possible to solve it analytically, yielding the quantities of interest as mathematical functions over some other quantity such as time. While analytical solutions are highly precise and are straightforward to examine, actually determining the solution can be very difficult or even impossible for complex models. In the case of the n-body problem, it is possible to analytically solve the trajectories in some simple specific cases, such as the case with only two bodies, but impossible in the completely general case.

The main analysis method of interest to us is simulation. Simulation means using numerical inputs to directly execute the model without solving it first. Compared to measuring real systems, while less accurate, simulation has several advantages. Any quantity within the simulation can be measured relatively easily, and it is feasible to freely experiment and tinker with the model and simulation parameters. Furthermore, a simulation can often run significantly faster than real-time, allowing simulations of extended time pe-

riods to run in a short time. Compared to analytic solutions, simulating is often easier, at the cost of accuracy, although both approaches are limited by the accuracy of the model being analyzed. Simulating the n-body problem could be done by numerically integrating the differential equations that describe the trajectories of the bodies.

4.2 Simulation

Simulations, where the simulation state changes only at specific times, called *events*, are called *discrete event simulations*. [2] Discrete event simulation can be used to simulate a very wide variety of systems, including network protocols and queuing systems.

On the other hand, if the simulation state contains continuous quantities (i.e. the state is changing all the time), the corresponding simulations are called *continuous simulations*. Commonly, continuous simulation involves simulating systems described using differential equations, such as fluid dynamics or electronic circuits.

There are also simulations where the passage of time does not play a significant role. Such simulations are called *static simulations*. One example of static simulation is *Monte Carlo simulation* [22] being used to numerically evaluate integrals that are very hard or impossible to solve analytically.

The most crucial part of a simulation is the simulation model, which describes how the simulation behaves at the desired level of detail. Choosing the appropriate level of detail often involves trade-offs involving model complexity, simulation time and result accuracy.

As an example, we look briefly at simulating a microprocessor, something which is commonly done to validate a microprocessor design before physical manufacturing.

A very detailed model could describe the actual transistors in terms differential equations, which could then be numerically solved. This kind of model can be very accurate, and could be used to study problems related to analog signal behavior inside the microprocessor, such as crosstalk. However, simulating this sort of model is computationally extremely heavy and nowhere near real-time.

A slightly more abstract model could describe the microprocessor at the so-called *register-transfer level* (RTL), where the microprocessor is modeled in terms of abstract digital signals (as opposed to analog in the transistor model above) and the logical operations performed on them. Simulating this sort of model is much faster than analog simulation, and the simulation can still be used to verify the microprocessor correctness and study its behavior

at the logical level. This kind of model could still be cycle-accurate (i.e. the simulated microprocessor uses an equivalent amount of simulated cycles as the real microprocessor would use clock cycles, even though what happens inside a single cycle has been partly abstracted), in which case it could be used for estimating and analyzing the performance of short programs (e.g. inner loops) executed by the microprocessor. Cycle-accurate simulation is usually still not real-time, and often too slow to simulate long runs of complex real programs like operating systems, although real-time simulation can be possible if the simulation system is very powerful and the simulated microprocessor very simple.

An even more abstract model could model whole subsystems (e.g. decoders, arithmetic-logic units, caches, etc.) of the microprocessor at a functional level, forgoing cycle accuracy for more simulation speed. This enables simulation of more complex programs such as entire operating systems running processes. This kind of model could be used, for example, to study cache behavior of programs.

In addition to the model, a simulation needs input data, called the *workload*. We can obtain a workload using several methods.

If a real system is available, we can observe the workload used by the real system and use that as simulation input. We can then use such a workload to validate that the simulation is working correctly by comparing the simulation results to those produced by the real system. When the simulation is working correctly, analyzing the simulation output can provide insight into the operation of the system. Furthermore, it might be difficult or even impossible to measure that kind of data from a real system.

We can also generate the workload randomly, by drawing the different inputs from various probability distributions. As this doesn't require detailed observations of the real inputs, it is much easier to implement than recording real workloads. If we can choose a distribution that describes the real inputs well, a reasonable assumption for many inputs, the simulation can also produce accurate results.

If the simulation has been verified and validated, generated inputs can also be used to explore the input space and study how the simulated system works in extreme scenarios or edge cases.

4.3 Simulators

A *simulator* is a computer program for doing simulation, either by means of a custom simulation package or by writing the simulation in a general-purpose programming language.

Simulators need to be correct, so their results can be relied upon. We need to *verify* that our simulator faithfully implements the simulation model, and we need to *validate* that our model adequately represents reality. It is said that verification is asking the question "are we building the thing right?" whereas validation is asking the question "are we building the right thing?". [29]

Another important property of simulators is the relation of simulated time to real time. Many simulators need to be able to run simulations faster than real-time, so that phenomena that would normally take weeks or years can be studied in shorter time periods. On the other hand, some phenomena in the real world happen so fast that it's hard to observe the details of what is happening. When simulating those phenomena, it is useful to simulate in *expanded time*, which can be thought of as simulating in slow motion.

Some simulation problems, such as those involving many complex differential equations, might be desirable to be simulated in real-time, but cannot because it is computationally hard or intractable to simulate it fast enough. With these kinds of problems, performance optimization of the simulator is very important.

4.3.1 Programming model

To use a simulator, we describe our simulation model to the simulator using the simulator's *programming model*. Depending on the simulator, the model can be defined by using a graphical user interface or by writing it using either a special simulation language or a general purpose programming language. All approaches have different benefits and trade-offs.

Using a GUI for modeling is easier for non-programmers to do, but can be inconvenient for complex models. Special simulation languages often have convenient simulation-specific features to ease modeling, but sometimes lack the expressive power and abstraction mechanisms available in general purpose languages. In addition, GUIs and special languages usually have worse runtime performance than general purpose languages. General purpose languages offer the best abstraction capabilities and performance, at the cost of a more complex implementation.

The more complex special simulation languages become, the closer they get to general purpose languages. Indeed, Simula [7], an early simulation language, had a strong influence in the development of modern object-oriented languages like C++ and later Java.

Many modern simulation packages use a general purpose language, often augmented with package-specific libraries or tools, for modeling. For example, CSIM [30], a fast and widely used simulation package, uses the C

programming language, while the NS-3 network simulator [13] uses the C++ and Python programming languages.

4.3.2 Discrete event simulators

Discrete event simulators [2] advance time (and thus the simulation state) only at certain times, either using a fixed timestep or using next-event time advance. A *fixed timestep* means that the simulator advances the simulation time by a constant in each simulation cycle. *Next-event time advance* means that the simulator, upon finishing a simulation cycle, determines the event that will occur next, and advances the simulation time to the time of that event. Since there are, by definition, no events in between and since only events change the simulation time, we don't miss any state changes by advancing directly to the next event. In practice, most discrete event simulators use the next-event approach, but a fixed timestep can still be useful in e.g. CPU simulation, where the timestep can be set to a single clock cycle of the CPU being simulated.

Commonly, the next-event time advance mechanism is implemented using a priority queue. All events in the simulation are timestamped and then put into a priority queue. After each simulation cycle, the simulator removes the first event from the priority queue and sets the simulation time to be equal to the time of the removed event. The new time can be the same as the previous time, which allows instantaneous events and reactions to occur. These kinds of simulation cycles where new events occur but time does not advance, are sometimes called *delta cycles*, especially in the context of hardware simulation.

In addition to the time advance mechanism, the simulation logic in discrete event simulators is usually organized in one of two ways: the *event-scheduling approach* or *process-oriented approach*.

In the event scheduling approach, the simulation logic resides in event processing code. The simulator determines the next event to run, and then runs the event handling code, which modifies the simulation state. The advantage is that the event scheduling approach is easy to implement very efficiently, but at the cost of making implementing complex models difficult.

In the process-oriented approach, the simulation logic resides in the entities that comprise the simulation. Instead of the events directly altering the state, the processes can react to occurring events and alter the state based upon that. In general purpose languages, these processes are often implemented using objects and/or threads, while specialized simulation languages might offer first-class constructs for them. The advantage of the process-oriented approach is that it makes complex models easier to understand,

implement, modify and add upon. However, using it requires either special language or library support.

Additionally, process-oriented simulators need to have a mechanism for processes to react to events. Two common options are broadcast events and event ports. In a *broadcast event* system, each event can be observed by every process, while in an *event port* system, events are sent to designated ports, and can be observed only by processes that are attached to those ports. A broadcast system is more flexible and doesn't require coupling between event senders and recipients, but broadcast systems are harder to implement efficiently.

Chapter 5

The RCOEX simulator

In this chapter, we introduce the Radio Coexistence (RCOEX) simulator that we implemented and used for studying the in-device radio coexistence problem. We present the design goals, a general implementation overview and a more detailed look at some particular features of the RCOEX simulator.

5.1 Design goals

To guide us through the design decisions of the RCOEX simulator, we set down three major design goals that the simulator should be able to meet: ease of modeling, interactive performance and reproducibility of results.

Ease of modeling means that writing (i.e. programming) radio simulation models should be easy and straightforward. Ideally, the programmer should be able to concentrate on the model itself and not generic programming concerns like memory management and configurability.

Interactive performance is required to be able to tweak simulation settings rapidly once the models have been developed. Good performance also facilitates running simulation batches with a significant amount of runs, which can help with the statistical significance of results.

Reproducibility of results means both determinism and easy installation and use. A deterministic simulator makes it possible to reliably reproduce interesting results once they are obtained, while easy installation and use make it possible for people other than the simulation authors to reproduce simulation experiments in their entirety.

5.2 Implementation overview

The RCOEX simulator is a process-oriented discrete-event simulator that uses next-event time advance and broadcast events. It is implemented using the Java programming language[26], which is also its programming model (i.e. the simulation models are implemented using Java, and are compiled along with the simulator core). This is similar to the approach taken in [9].

The RCOEX simulator is composed of two generic subsystems, each of which we will discuss in more detail in subsequent sections:

Configuration and launcher , which initializes the simulator upon startup.

Tasks and events , which the simulation model is programmed with.

5.3 Configuration and launcher

The configuration and launcher subsystem is responsible for initializing the simulator into the desired state from scratch. It accepts both command line parameters and configuration files, mostly in the form of *configuration variable assignments*, which are hierarchical dot-separated names followed by an equals sign and the desired value, e.g. `wlan0.atim-period=1000ms` or `lte0.drx.offset=0`. In addition to configuration variable assignments, command line parameters are also used for specifying which models are included in the simulation. For example, the command line could contain `--proto=wlan,wlan0` to add a WLAN model with the name `wlan0` to the simulation.

The configuration subsystem uses reflection to make it easy for simulation models to allow configuration with only a minimal amount of code. Listing 5.3 presents a minimal example of a model class that can be configured. The model in question just has to implement the `Configurable` interface by implementing the `getConfiguration` method. Then, at runtime, the simulator core will update the object returned by that method, called the *configuration object*, to contain the active configuration values, which the model can use by just accessing the fields of the object directly.

The configuration object itself is just a very simple data container class with annotated public fields for each configurable variable. Each public field that has a `Description` annotation and/or a `VariableDefinition` annotation can be configured using a configuration variable assignment. The `Description` is used for defining a runtime help text, while also serving as a comment. The `VariableDefinition` can be used for renaming the variable using the `id` parameter or specifying the physical units accepted.

```
public class ConfigurableModel implements Configurable {
    private ModelConf conf = new ModelConf();

    public Object getConfiguration() {
        return conf;
    }
}

public class ModelConf {
    @Description("Configurable duration value")
    @VariableDefinition(id = "duration", unit = Time.class)
    public long durationNs = 1000;
}
```

Figure 5.1: Minimal configurable model

5.4 Tasks and events

The task and event system is the part of the RCOEX simulator that handles the simulation processes which form the simulation model. The model is composed of *tasks*, which are autonomous simulation entities that are able to send and receive *events*. The tasks interact with the simulation using programming primitives provided by the simulator core.

Tasks are defined by extending the `Task` class and overriding the `simulate` method. The `simulate` method starts to run when the task is started, which can be $t = 0$ if the task exists at the beginning of the simulation.

There are four main primitives available for tasks to interact with the simulation. For waiting, tasks can use `delay` and `waitUntil`. For firing and receiving events, tasks can use `fireEvent` and `receive`.

5.4.1 Waiting primitives

The RCOEX simulator supports two kinds of waiting primitives, which allow the simulation tasks to model time passing in the simulation. `delay` is used to wait for a specific duration and `waitUntil` is used to wait until a specific moment in simulated time.

Listing 5.2 demonstrates the use of the waiting primitives and `fireEvent`. The example task first waits until the simulation time is exactly $t = 100$ ms, fires an event, waits for 1 ms (the simulation time at this point is therefore $t = 101$ ms) and then fires another event.

```
public class DelayWait extends Task {
    public void simulate() {
        // Stops the task simulation until the simulation
        // time is exactly 100 ms.
        waitUntil(Time.ms(100));

        // The simulation time is now 100 ms.

        fireEvent(new ExampleEvent());

        // Stops the task simulation for a duration of
        // 1 ms of simulated time.
        delay(Time.ms(1));

        // The simulation time is now 100 ms.

        fireEvent(new ExampleEvent());
    }
}
```

Figure 5.2: `delay`, `waitUntil` and `fireEvent`

5.4.2 Event firing and reception

The event system of the RCOEX simulation is built upon two primitives, `fireEvent` and `receive`. Of these, `fireEvent` is extremely simple: given a simulation event, which can be any Java object that inherits the `Event` class, it makes that simulation event occur at the current simulation time. The events are broadcast to all tasks in the simulation.

The most interesting part of the task system is `receive`, which tasks use for reacting to events. It is based on Erlang's receive statement, and uses a *fluent API* [32] to offer a convenient interface that is able to benefit from tooling such as type-aware automatic completion.

The `receive` construct has three major features: *filtering*, *timeouts* and *pattern matching*. Filtering enables tasks to declare which events they are interested in. Timeouts enable tasks to stop reacting if no suitable event occurs within the desired timeframe. Pattern matching allows tasks to declare several filtering criteria simultaneously.

When we want to receive events, we first call `receive` to obtain a *receiving context*, which defines the time scope of the events we are interested in. Only those events that occur after the creation of the receiving context are

```
// Accept any event.
Event e = receive().matchAny().get();

// Accept only events of type SomeEvent.
SomeEvent e = receive().eventType(SomeEvent.class).get();

// Accept only events of type PHYEvent that
// are related to the given PHY "somePhy".
PHYEvent e = receive().match(PHYEvent.phy, somePhy).get();

// Accept only events of type PHYEvent that
// are related to the given PHY "somePhy" and
// are also RX events.
PHYEvent e = receive().match(PHYEvent.phy, somePhy)
                      .match(PHYEvent.isRX, true)
                      .get();

// Accept only events of type NumEvent
// that return 123 from their .num() method.
NumEvent e = receive()
    .eventType(NumEvent.class)
    .when(new Predicate<NumEvent>() {
        public boolean apply(NumEvent e) {
            return e.num() == 123;
        })
    .get();
```

Figure 5.3: Filtering with `receive`

considered, so events in the past are not received.

Using the receiving context, we can define any amount of criteria using one or more of the `matchAny`, `eventType`, `match`, and `when` methods. Finally, we can call the `get` method, which halts the task until the specified event occurs and then returns that event object or `null` if a timeout occurs. Listing 5.3 contains several filtering examples.

`receive` utilizes the available static type information so that if we have specified that only events of type `SomeEvent` should be accepted, the static return type of `get` is also `SomeEvent`, relieving us of superfluous type casts.

Sometimes, it is convenient to have several alternative criteria for incoming events. For example, we might like react differently based on whether event x or event y occurs first. `receive` supports this with pattern matching.

```
// obtain a receiving context
Receiver r = receive();

// match a SomeEvent that has an intVal of 123 and
// a boolVal of false
for (SomeEvent e : r.match(SomeEvent.intVal, 123)
                    .match(SomeEvent.boolVal, false)) {
    doSomething(e);
}

// otherwise, match any SomeEvent
// this does NOT match if the above matched!
for (SomeEvent e : r.eventType(SomeEvent.class)) {
    doSomethingElse(e);
}

// it wasn't an IntEvent, match an OtherEvent instead
for (OtherEvent e : r.eventType(OtherEvent.class)) {
    andNowForSomethingCompletelyDifferent(e);
}

// done with these alternatives, obtain a new context
r = r.again();
```

Figure 5.4: Pattern matching

Listing 5.4 presents an example of pattern matching. First, we call `receive` to obtain the receiving context, which we store in the variable `r`. Then, we can use that receiving context together with the desired criteria in a Java for-each loop to match on those criteria and bind the matching event in a variable. To define several alternatives, we can use several for-each loops. We say that each of these for-loops is a pattern, and is read as "for these kinds of events, do this". Once we are done with the patterns, we can use `again` to obtain a fresh receiving context to receive more events.

The patterns share the receiving context, which makes it possible for them to be mutually exclusive. As soon as a pattern matches and the code inside the for-statement is executed, all subsequent patterns automatically fail to match. This is similar to Erlang, where the first matching pattern takes precedence.

```
// when using .get(), a timeout returns a null
AnEvent e = receive().timeoutAfter(Time.ms(50))
                     .eventType(AnEvent.class)
                     .get();

if (e == null) {
    // timed out!
}
else {
    doSomething();
}

// when using pattern matching, timeouts can
// be checked with .timedOut()
Receiver r = receive().timeoutAt(Time.ms(10));

for (AnEvent e : r.eventType(AnEvent.class)) {
    doSomething();
}

if (r.timedOut()) {
    // timed out!
}
```

Figure 5.5: Timeouts

5.4.3 Reception timeouts

We are frequently interested in some event only if it occurs during some time period. `receive` supports this using the timeout feature, which can be used via the methods `timeoutAfter`, `timeoutAt`, `neverTimeout`, and `dontBlock`. `neverTimeout` is the default and also the simplest: it disables timeouts, so the task always waits until a matching event occurs. `timeoutAfter` specifies a relative timeout, where only events that occur within the given duration, starting from the current simulation time, are considered. `timeoutAt` is the absolute counterpart, where only events that occur before the given simulation time are considered. `dontBlock` allows the task to poll without blocking (`get` always returns instantly) for events that have already occurred during the existence of the receiving context. As such, it can be considered a relative timeout of zero. Listing 5.5 contains several examples of timeout use.

```
try {
    // enable interrupts
    interruptibleBy(EventMatcher.matcher()
        .eventType(WindowExpired.class)
        .predicate());

    // the following code is automatically interrupted when
    // a WindowExpired event occurs
    while (true) {
        if (carrierSense()) {
            delay(waitDuration);
        }
        else {
            attemptTransmission();
            waitForAcknowledgement();
        }
    }
}
catch (Interrupt i) {
    // WindowExpired occurred
    waitForNextWindow();
}
finally {
    // disable interrupts again
    interruptibleBy(null);
}
```

Figure 5.6: Interrupts

5.4.4 Interrupts

The simulator also supports *interrupts*, which allow a task to specify that certain events will interrupt any ongoing waiting, such as `delay` or waiting for an event using *receive*, and cause an `Interrupt` exception to be thrown. As always when using exceptions, interrupts are intended to be used for exceptional cases only, where the equivalent interrupt-less code would be needlessly convoluted.

Listing 5.6 presents an example of using interrupts. We can use the primitive `interruptibleBy` along with a boolean predicate to specify the exceptions that cause an interrupt, and later `interruptibleBy` with `null` to disable the interrupt mechanism.

5.4.5 Implementation

The heart of the task and event system is the *task scheduler*, which is implemented using a next-event time advance mechanism, which in turn uses a conventional priority queue approach. The task scheduler is responsible for selecting which simulation task is run and when, and for broadcasting events to tasks.

5.4.5.1 Tasks and threads

In the RCOEX simulator, tasks are implemented using threads. The task scheduler and all tasks run in their own threads and contain a dedicated semaphore. Initially, the scheduler is executing and each task thread is blocked on its semaphore. When `resume` is called, the scheduler releases the semaphore of that task and immediately blocks on the scheduler semaphore. When `suspend` is called later, the task thread releases the scheduler semaphore and immediately blocks on its own semaphore.

The semaphore scheme means that, although there are several active threads in the simulator, only one thread is ever executing at a time. Thus the RCOEX simulator is logically single-threaded, even though it is implemented using several threads. Threads are used only for having a separate call stack for each task and not for parallel execution. Additionally, the RCOEX simulator always exactly controls which thread is executing at any given time, which means that no nondeterminism is introduced because of thread scheduling.

Each task also has its own *mailbox*, which is a simple queue of events.

5.4.5.2 Simulation cycles

After initialization, the RCOEX simulator runs *simulation cycles* in a loop. Each simulation cycle is a single run of the task scheduler. A cycle always advances the state of the simulation, either by advancing the simulation time (if there were no events) or by resuming tasks in reaction to events.

The task scheduler has a few key data structures that it uses. The *ready list* contains all tasks that will be run (i.e. resumed) during this cycle. The *new event list* contains all events that were fired by tasks during this cycle. The *wakeup queue* contains all suspended tasks in ascending time order such that the task to wake up soonest is the first in the queue. The *waiting set* contains all tasks that need to be resumed whenever an event occurs, regardless of the wakeup time, each with an optional *wakeup predicate*. Similarly, the *interruptible set* contains all tasks that have an *interrupt predicate* set.

At the start of each simulation cycle, the scheduler goes through every task in the ready list and calls `resume` for each task, which enables the model code of that task to immediately execute. During task execution, the scheduler collects all events fired using `fireEvent` into the new event list. Eventually, one of the following happens:

- The task throws an exception, which causes the simulation to be immediately terminated with an error message.
- The task code terminates, in which case the task is removed from the simulation.
- The task calls `suspend`, which returns the control to the scheduler.

After a task calls `suspend`, if it set a wakeup time, it is put into the wakeup queue. Additionally, the task might have placed itself into the *waiting set* and/or the *interruptible set* to react to events. At any given time, the scheduler upholds the invariant that all tasks must reside in either the ready list or one or more of the wakeup queue, the waiting set, and the interruptible set. A task that is in the ready list is never in any of the other data structures and vice versa.

After all tasks in the ready list have been run, the ready list is empty and the scheduler goes through the new event list. For each event, the scheduler goes through the waiting set and the interruptible set. If a task is in the waiting set and it either has no wakeup predicate or its wakeup predicate returns true for the event in question, the scheduler wakes it up, puts the event in its mailbox and places it into the ready list. If a task is in the interruptible set and its interrupt predicate returns true for the event in question, that task is *interrupted*, which means that it is put into the ready list and an `Interrupt` exception is thrown when the task resumes execution. Additionally, all events in the new event list are outputted into the global event trace.

Finally, the scheduler determines the new simulation time for the next simulation cycle. If there are events in the ready list, which is the case if an event woke up a task that was waiting, the new time is the same as the current time. Otherwise, the wakeup queue is examined. The first task (i.e. the task with the nearest wakeup time) in the wakeup queue is extracted, put into the ready list and the new simulation time is set to be equal to the wakeup time of that task. If there is no such task, which means that the wakeup queue was empty, the simulation terminates as there is nothing left to run.

5.4.5.3 Primitive implementation

In the simulator core, `delay`, `waitUntil`, and `receive` are implemented using the low-level primitives `suspend`, `resume`, `waitForEvent`, and `acceptsOnly`. These low-level primitives are below the level used by the user-written simulation models, and are not used by the models directly. Additionally, there is the `interruptibleBy` primitive, which is used by tasks directly.

`resume` transfers the control from the scheduler to a task, allowing it to execute arbitrary code until it voluntarily yields execution back to the scheduler using `suspend`. Before calling `suspend`, the task implementation can optionally set a *wakeup time* by setting the `wakeUpAt` variable. `delay` and `waitUntil` are implemented by simply computing the absolute wakeup time and setting the `wakeUpAt` variable to that value.

`waitForEvent` is like `suspend`, except that the task is also put in the waiting set. To implement `receive`, the code calls `waitForEvent` and upon waking up, checks that the received event (which is taken from the task's mailbox) matches the `receive` criteria. Optionally, as a performance optimization, the task can also set its wakeup predicate using `acceptsOnly` before calling `waitForEvent`, which enables the scheduler to check for valid events without a context switch. The `receive` implementation does this whenever it knows the entire set of valid events, which is whenever `get` is used. However, `acceptsOnly` cannot be used when `receive` pattern matching is used.

`interruptibleBy` is exactly like `waitForEvent`, except that the task is put in the interruptible set instead of the waiting set, and the interrupt predicate is mandatory.

Chapter 6

Simulating in-device radio coexistence

After developing the RCOEX simulator as described in the previous chapter, we used it to study the previously introduced problem of in-device radio coexistence (IDC). In this chapter we describe how we used the simulator to perform our simulations.

6.1 Coexistence cases

As our subjects of study, we selected two particular *coexistence cases* that we deemed especially valuable to solve.

Both of our studied cases were related to interference between LTE and WLAN. As introduced in chapter 3, this can occur when LTE and WLAN symmetrically interfere each other near LTE TDD band 40 and when LTE transmitters on LTE FDD band 7 interferes WLAN receivers. We identified two particular real-world cases where this is likely to happen, if the device is operating on the bands mentioned above.

Mobile hotspot The device is connected to the Internet using LTE, and shares its Internet connection with nearby devices by acting as a WLAN access point. Because traffic from one connection is usually routed to the other, the traffic patterns of both connections are very similar to each other. If the device suffers from in-device interference, hotspot users may suffer from bad connectivity.

WLAN off-loading The device is simultaneously using its LTE and WLAN connections. For example, the device may wish to route low-latency traffic, such as VoIP, through LTE, which offers stronger guarantees on

when and how much the device is able to transmit. However, it might make sense to the device to route non-critical high-bandwidth traffic, such as streamed video, through WLAN, which might not carry usage charges for high data amounts. LTE-to-WLAN interference in this case can severely degrade WLAN throughput and lead to much spectrum being wasted, hurting other possible users of the WLAN network.

6.2 Models

6.2.1 Radio models

In order to study these cases in detail, we built detailed protocol-level simulation models of both LTE and WLAN. Protocol-level means that we simulate on the level of protocol packets, and abstract lower level concepts such as radio symbols and analog signals.

The LTE model includes both time-division duplex and frequency-division duplex modes, as well as realistic framing with different TDD subframe configurations, HARQ processes and DRX support. The model has separate support for both eNodeB and UE.

The WLAN model includes beacons, power-saving mode support and realistic link adaptation in the face of interference. Like actual WLAN, it uses CSMA/CA with random backoff.

Both LTE and WLAN models support MIMO operation, allowing them to utilize several RX and TX radio pipes at once. We will define pipes in subsection 6.3.1.1 below.

6.2.2 Interference model

To model the coexistence cases introduced in the beginning of this chapter, we built an abstract switchboard-like interference model. It allows connecting any two radio pipes using a one-way connection that causes interference in the other pipe whenever the other pipe is transmitting (in the case of TX pipes) or receiving (in the case of RX pipes).

Because the connection is one-way, this easily allows modeling of asymmetric interference. In addition, the model allows configuring RX pipes to interfere TX pipes. While this doesn't seem sensible from a physical standpoint, it can be used to model real-world priority override scenarios where the transmitter of a low-priority radio, such as WLAN, is hard-wired to be powered off whenever a high-priority radio, such as LTE, needs to be able to receive transmissions.

Finally, although real-world interference is not absolute in the sense that interfered transmissions might still succeed, our abstract model is conservative and causes all interfered transmissions to fail entirely. If the developed IDC mitigation techniques work well even using this conservative approximation, their real-world performance should hopefully be more than adequate.

6.3 Implementation

The most important parts of our simulation system are of course the radio models, the important features of which were introduced above. The radio models are responsible for deciding when and what to transmit and receive, which in turn leads to those transmissions either succeeding or failing, based on interference conditions.

In addition to the radio models and the interference model, the system contains a simple scripted workload generator. While simple, it was adequate to model the coexistence cases introduced above.

Finally, our simulation also contains tracing and analysis subsystems, which graphically visualize the simulation results, allow interactive inspection of the simulated timeline, and calculate statistics such as throughput and packet loss.

6.3.1 Implementation overview

As described in chapter 5, our simulator is process-oriented. The radio models, workload generators, interference managers and tracing and analysis systems are implemented as autonomous simulation processes that communicate mostly by firing and receiving events.

6.3.1.1 Radio models and pipes

Each radio model has associated *pipes* for which it generates and receives *radio events*. A pipe is a representation of a radio resource which is *unidirectional* (i.e. it can either send or receive transmissions, but not both) and *atomic* (i.e. it can't be subdivided further). As such, a pipe is roughly analogous to a physical radio antenna. However, a physical antenna can be bidirectional, which would be modeled as an RX/TX pair of pipes in our model. A pipe can be in a number of distinct states. Transitions between the states are marked by by radio events. The allowed states are as follows:

Powered A pipe can be either powered on or powered off, which represents the physical power state of the power amplifier connected to the

antenna.

Transceiving A pipe can be either idle or transceiving, which represents whether meaningful traffic is being transmitted through it. We say that transceiving RX pipes are receiving and that transceiving TX pipes are transmitting. Pipes that are powered off cannot transceive, which means that a transceiving state implies that the pipe is also powered on.

Interfered A pipe can either be uninterfered or interfered, which represents whether the pipe is receiving so much interference that no transmissions through it will succeed. We generally assume that pipes are uninterfered, and say that a pipe is being interfered when it is in the interfered state.

Each state has a corresponding on/off pair of radio events, which are directly implemented as simulation events. Whenever a pipe moves from one state to another, the corresponding events are fired using `fireEvent`. These events completely describe the physical behavior of the simulated radio interfaces, and are visualized by our tracing and analysis system.

6.3.1.2 Layer 2 events

The low-level radio events allow determining if a particular transmission succeeded or not, and its duration in simulated time, but they are not enough on their own. For example, determining throughput requires knowledge of the modulation being used (i.e. amount of bytes per second) and the amount of protocol overhead. To enable these kinds of statistics to be computed, our system uses the concept of *frames*. Frames are the unit of work that the workload generators generate; each frame is simply defined as a total of bytes. Each radio model is responsible for determining how frames map to actual physical transmissions and for determining when and if they are completely transmitted. Then, accordingly, the radio models fire following kinds of *layer 2 events* to enable analyzing high-level information:

Physical A physical transmission has finished. The event contains additional status information, such as amount of bytes transmitted and whether the transmission succeeded or failed.

New A new frame has been accepted for transmission or reception, and will be simulated by the radio model.

Complete A frame has successfully completed, which means that the receiver has successfully received all bytes in that frame.

Aborted The frame transmission has failed, and all bytes in the frame were lost.

The tracing and analysis subsystem utilizes the layer 2 events to compute many useful statistics for each radio model. The statistics include the following:

Protocol-level throughput How many bytes were successfully transmitted per second by each protocol?

Packet loss How many and what percentage of frames were lost due to interference?

Latency distribution How long did frames take to successfully complete?

These computed statistics were used to evaluate the severity of different coexistence cases and the effectiveness of developed mitigation techniques.

6.3.2 Abstract protocol and PHY

Even though the high-level logic is vastly different for our radio models, all radio models nevertheless have some common characteristics. We have implemented these common characteristics as an *abstract protocol*, an abstract class which the actual radio model implementations inherit from.

The most obvious common characteristic is that all our radio models have a *virtual PHY*, short for virtual physical interface, which is a collection of RX and TX pipes. A SISO model will have at most one each, while a MIMO model can have several. The abstract protocol includes a virtual PHY and offers a number of utility methods to easily manipulate the associated pipes.

In addition to a virtual PHY, all radio models process workload frames, so the abstract protocol also includes helper methods to automatically produce the proper layer 2 events when workload frames are queued for transmission or reception and when they succeed or fail.

6.3.3 Workload generation

Our workload generator subsystem is a very simple scripted system. As a part of the simulator configuration, the user can add one or several workload generators into the simulation. Each workload generator has a configurable starting time (the time of the first transmission) and an optional repeat interval, after which it will periodically repeat the transmission. The amount and size of RX and TX packets per transmission is also configurable.

While this system is very simple, it is possible, if somewhat arduous, to build arbitrarily complex predetermined workloads out of these generators, as individual packets can be scripted using single-packet workload generators. Fortunately, we found that simple arrangements were enough for analyzing our coexistence cases.

6.3.4 Interference caster

Our simulator supports two kinds of interference: in-device interference between different radios in the simulated device and scripted external interference.

Scripted interference works similarly to workload generation. As a part of simulator configuration, the user can add any number of single interference events and periodic interference events that affect some radio model in the simulation. If interference that affects several radio models at once is desired, it can be modeled as simultaneous single-radio interference events.

Perhaps more interesting is the in-device interference system. As described in subsection 6.2.2, the user can configure that the RX or TX of some radio model *casts* interference to the RX or TX of some other radio model. MIMO RX and TX are considered as a single unit for the purposes of in-device interference modeling.

The interference casting system allows easily modeling actual transmission interference, e.g. when the TX of LTE interferes the RX of WLAN, by configuring the LTE TX to interfere WLAN RX. In-device interference because of transmission priorities, e.g. WLAN TX being powered off when LTE needs to receive, is also straightforward to model by configuring the LTE RX to interfere WLAN TX. The system supports an arbitrary number of these kinds of configurations.

6.3.5 Tracing and analysis

Figure 6.1 shows a screenshot of our trace visualizer GUI. The visualizer displays a left-to-right timeline, where all TX and RX pipes in the simulation are shown on different lines. The pipes are labeled in the left side of the window.

The differently colored sections in the timeline correspond to different states a pipe can be in. A transparent section (i.e. no bar at all) means that the pipe is completely powered off at that time. A green section means that the pipe is powered on, but not actively transmitting or receiving. A black section means that the pipe is either transmitting (if it is a TX pipe) or receiving (if it is an RX pipe). The endings of transmissions are marked

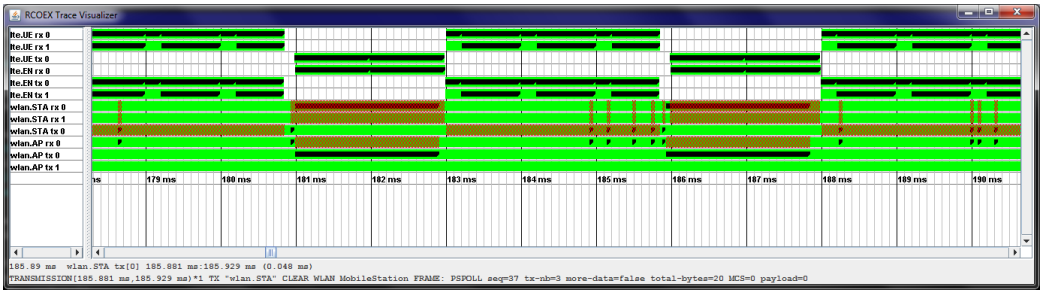

Figure 6.1: Trace visualizer

with small notches in the black bars, so that consecutive transmissions can be distinguished. Finally, a red section means that the pipe is interfered. For RX pipes, this means actual radio interference, while for TX pipes this means that the pipe is being overridden by a more high-priority radio.

In the example in figure 6.1, the sources of the displayed interference are straightforward to see. Whenever WLAN is transmitting, it interferes WLAN RX in the same device. Also, whenever LTE is receiving in the UE, the WLAN TX of the device is overridden.

In addition to the bars, the visualizer offers more detailed information about the individual transmissions. When the mouse cursor (not shown in figure 6.1) is hovered over a transmission (black bar), protocol-level details of that transmission appear in the lower text area. In figure 6.1, the transmission in question is shown to be a PS-POLL packet.

To be able to obtain numerical results that can be easily compared between different simulations, our system also computes some statistics about the simulated workload. These statistics include:

- Effective TX and RX throughput of all protocols in the simulation
- Total amount of transmissions
- Amount of failed transmissions
- Total seconds transmitted
- Total seconds that were spent transmitting failed transmissions
- Amount of payload bytes successfully transmitted
- Amount of payload bytes lost due to failed transmissions
- Total and failed amounts of various low-level WLAN frames

- Latency histogram of transmitted payloads

When comparing different approaches for in-device interference mitigation, such as our implemented coexistence strategies, these statistics can be used to quantitatively support one approach over another.

Chapter 7

Discussion

In this chapter, we discuss, review and evaluate the results of the RCOEX simulator and coexistence simulation development. In particular, we attempt to answer the following questions:

- Did we meet our design goals (as given in section 5.1) for the simulator?
- Was the simulator development completed within the allotted time?
- What simulator features proved useful and successful?
- What simulator features were unnecessary or lacking? Where a feature was found lacking, how could it be improved?
- Was the simulator used successfully for in-device coexistence research?

7.1 Design goals

In section 5.1 we set the following three main design goals for the simulator: ease of simulation development, adequate performance for interactive use and easy reproducibility of results. We review each of these goals individually below.

7.1.1 Ease of simulation development

Difficulty is hard to objectively measure, especially when implementing highly complex systems, such as the LTE model. However, we can attempt to make some observations.

All models were completed in time, and without having to fight around limitations in the simulator. The WLAN model in particular benefited much

from the some of the more specialized simulator features, such as interrupts, simplifying the implementation significantly.

However, the LTE model, which is by far the most complex model developed in the project, did not end up using many of the simulator's event related features. Instead, the LTE model communicates internally using custom objects, shared state and direct method calls, using the simulator features only for time delays and firing tracing events. From the features introduced in chapter 5, the LTE model uses only `delay` and `waitUntil` directly. In addition, `fireEvent` is used by the pipe simulation mechanism which the LTE model also uses.

While it appears that the simulator did not make the implementation of the LTE model more difficult, it did not make the implementation easier either. In this regard, we can say that the simulator failed to provide features that would have accelerated LTE model development. The sole exception to this is the configuration system, which certainly made it easier to tune the LTE configuration parameters.

Although it might not be the only reason, one possible explanation why the LTE model eschews event based communication is the complexity of the LTE protocol state, which has, among other things, several simultaneous HARQ processes and their associated timings (see section 2.2). If only events were used for LTE model communication, all protocol state changes would need to be encoded using event firing and reception (as opposed to directly mutating common state), which would have been time-consuming and unproductive.

Even in retrospect, it is hard to think of general features (i.e. not purely LTE-specific ones) that would have dramatically simplified the LTE model. LTE is a complex protocol: it should not be surprising that the model is also complex.

7.1.2 Performance

The simulations used in coexistence experimentation were predominantly only 1000ms long in simulated time. When run on a relatively modern desktop computer, these kinds of simulation runs usually completed within a few seconds of being started. We deemed such runtimes to be adequate for interactive use, where the user often visually analyzes the results for periods much longer than the actual runtime.

While the simulator is fast enough for interactive use, our benchmarks show it to be far too slow for computationally intensive batch usage, especially if long simulation runs are required. As an example, a fast simulator could be used to automatically tune simulation parameters using an opti-

Simulated time	Wall clock time	Ratio
1000 ms	4220 ms	0.237
5000 ms	20106 ms	0.249
10000 ms	51106 ms	0.196
20000 ms	141266 ms	0.142
60000 ms	N/A	N/A

Figure 7.1: Performance benchmarking results

mization algorithm such as simulated annealing. However, as such usage was not required by the project, we did not find it necessary to do dedicated performance optimization.

7.1.2.1 Benchmarks

Table 7.1 shows the results of 5 performance benchmarks of the simulator. All benchmarks were run on the author's laptop computer (Intel Core i7-740QM (Clarksfield), 1.73 GHz, 6 GB RAM). It should be noted that even though the simulator is implemented using multiple threads and a multi-core CPU was used for benchmarking, the simulator uses only one of the cores, as only one of the threads is ever in execution at a time. All benchmarks were run with the same configuration: simultaneous LTE eNodeB, LTE UE, WLAN device and WLAN access point simulation with a periodic workload such that all radio pipes in the simulation had transmissions throughout the simulated duration. The benchmark configuration was one of the configurations that was used in the project to simulate WLAN throughput in the face of heavy in-device interference from LTE.

The first column of table 7.1 displays the simulated time durations for each benchmark. The second column shows the amount of wall clock time elapsed between the start and end of the actual simulation (simulator initialization, configuration, trace analysis and visualization are not included), measured using the timestamps in the simulator's log output. The third column shows the simulated time divided by the elapsed wall clock time, which measures whether the simulator is faster or slower than real-time.

As can be seen from the table, the simulator is noticeably slower than real-time, taking more than four times the simulated time to perform the simulation in the best case. Perhaps even more significantly, the simulator scales significantly worse than linearly with respect to the simulated time. In other words, doubling the simulated time more than doubles the amount of time it takes to run the simulation. The time and ratio for the 60000 ms run are displayed as "not available", because the benchmark run was aborted

after running for 20 minutes at full CPU utilization without terminating. We suspect this to result from either a non-termination bug in the simulator or a catastrophic degradation of performance.

While we did not investigate the reason for this worse-than-linear scaling behaviour, we can venture some educated guesses. Because the simulator continuously collects data (e.g. trace information) as it runs, a long simulation will need to store much more data than a short one, thus significantly increasing memory consumption. This can slow the simulator down in several different ways:

- As the simulator is continuously allocating new objects, garbage needs to be collected periodically. When there is a large amount of live objects, garbage collection can become slower.
- When the size of the simulator working set grows, it will eventually exceed the size of the CPU caches and possibly the TLB, causing performance to degrade. The CPU used for benchmarking contains three separate cache levels (L1, L2 and L3), so this kind of effect can happen multiple times. In theory, the working set could also exceed the amount of physical RAM, causing paging and further performance degradation, but this did not happen in our benchmarks.
- Any algorithms that are not $O(N)$ or better can start to become bottlenecks. We have not investigated whether such algorithms exist on the critical paths of the simulation.
- Although the simulator does not appear to leak large amounts of memory during simulation, it is possible that there are bugs causing needless objects to accumulate in simulator data structures, slowing down the processing of those data structures. For example, it is possible that some task in the simulation never flushes its event queue, causing event objects to accumulate. In this case, even if the event filtering algorithm is $O(N)$, it will start to become progressively slower as it needs to process more and more events.

7.1.2.2 Future optimization

If good performance for long simulation runs is required in the future, the reason for the performance problem visible in the benchmarks needs to be investigated and remedied. A good starting point for this kind of investigation could be to profile short and long simulation runs separately and attempt to discover which parts of the simulator take disproportionately longer in the long runs.

In addition, to improve the overall simulator performance, one promising optimization target could be the event reception and matching mechanism. The current implementation does attempt to avoid unnecessary context switching using `acceptsOnly` where possible, and in such cases performs the match in the scheduler thread instead of the task thread. However, the event matcher code still has to be run for each event and for all tasks that are waiting for events.

Currently, that event matcher code is implemented as a chain of virtual method calls, which are potentially slow to execute, unless the Java just-in-time compiler manages to devirtualize and inline them. If it is determined that the JIT fails to optimize the matching adequately, one possible optimization technique would be to manually emit Java bytecode that implements event matching with lowest possible overhead.

7.1.3 Easy reproducibility

Our results should be straightforward and easy to reproduce, both with regard to simulator installation and its use.

The simulator is written in the popular Java[26] programming language and uses the widespread Maven[33] packaging and build system. After Maven and Java are installed, the simulator (along with all our simulation models) should be buildable with a single `mvn package` command.

The entire simulation can be configured using plain-text configuration files, which describe the models used along with their parameters and workloads. The configuration files can also include a fixed random seed to reproduce any pseudorandomness in the simulation (e.g. pseudorandom workload generation), so simulation results obtained earlier can be reproduced exactly.

7.2 Implementation schedule

Once the initial design was in place, the development of the RCOEX simulator itself went smoothly and quickly, although some debugging effort was required to solve issues related to threading and exception handling. The quick initial development made it possible to spend a significant amount of the remaining implementation time tweaking the API according to feedback to make it as pleasant to use as possible in Java. All necessary features were completed in time for use in the actual protocol modeling and coexistence simulation.

7.3 Simulator features

Most of the implemented features proved useful in actual simulation development. Most notably, the configuration system (section 5.3) was used for every subsystem, greatly accelerating both model development and experimentation with the simulation. The event reception and filtering system (`receive()` and `match()`, subsection 5.4.2) was also widely used, and proved expressive enough to cover all encountered task communication needs. We regard these two features to be significantly successful.

Both interrupts (subsection 5.4.4) and timeouts (subsection 5.4.3) were used in WLAN model implementation, where they simplified the implementation of tricky protocol details. Even though the WLAN model was the only place where these features were used, we regard them as mildly successful as they made the code easier to develop and to read.

Pattern matching (listing 5.4) using `for` was not used in simulation model development. We believe that a significant reason for this was the limited use of events for simulation model communication. The use case for pattern matching arises when simulation tasks are receiving multiple events of different types simultaneously. However, our models do not use events of multiple types: tracing and WLAN mostly use events of a single type with varying data fields, and as discussed in subsection 7.1.1, LTE mostly does not use events at all.

When initially designing the simulator core, the decision to implement `for`-style pattern matching was based on our attempt to imitate Erlang's message passing as faithfully as possible in Java. We think that if the models had used more events, which was how we anticipated models to be written, pattern matching would have been much more useful than it was now. However, in retrospect, the pattern matching feature could have been omitted initially and then added later if it became necessary.

Chapter 8

Conclusions

8.1 Conclusions

In this thesis we first briefly reviewed some related radio concepts, radio protocols and simulation fundamentals in chapters 2 and 4. We also presented the problem of in-device radio coexistence along with strategies for managing it in chapter 3. We then presented our custom in-device radio interference simulator, the RCOEX simulator, with implementation details in chapter 5. The RCOEX simulator is a process-oriented discrete-event simulator implemented in and programmable using the Java programming language. We outlined how the simulator was used to simulate in-device radio coexistence in chapter 6 and discussed our results in chapter 7.

The main contribution of this thesis is the design and implementation of the simulator core of the RCOEX simulator. The simulator, which was developed in the project this thesis was a part of, was used to conduct research on in-device radio coexistence. In particular, it was used to experiment how radio protocols, such as 3GPP Release 8 (LTE) and the 802.11 family of wireless LAN protocols, interact with each other in in-device interference conditions.

During experimentation, the configuration and event filtering features of the RCOEX simulator, designed and implemented as part of this thesis and presented in chapter 5, proved very useful for simulation model development. We regard these features as the biggest successes of the design. Other features were also developed and presented, but they were less impactful for the research project on the whole.

The in-device radio coexistence research led to the development of coexistence strategies, which were measured to improve spectral efficiency and overall radio throughput in simulations. Even in the presence of harsh inter-

ference where overlapping transmissions always cause simultaneous transmissions to fail, much of the performance loss can be recovered if the radios can be made to co-operate. The coexistence strategies are presented with measurements in [18, 19]. In addition to these publications, two patents [20, 35] were filed. The RCOEX simulator presented in this thesis was instrumental in obtaining these results.

8.2 Future work

The RCOEX simulator suffers from known performance problems in simulation runs that span a long period of simulated time. These problems were discussed in chapter 7. By fixing these problems and optimizing the simulator implementation, longer simulation runs could be performed, and the simulator could potentially be feasibly used for automatically searching for good tuning parameter values for coexistence strategies. As it is now, the RCOEX simulator performance is low enough that such automatic searching would take too long to be very useful.

In addition to improvements on the simulator itself, the simulator could be used to experiment with protocol combinations other than just 3GPP Release 8 (LTE) and 802.11 (WLAN). Simulation models could be developed for radio protocols such as Bluetooth, GPS, and GLONASS. Simulation experiments could then be conducted to find working coexistence strategies for situations where some or all of these are operating simultaneously within a single mobile device.

Bibliography

[1] Baker, M. 3GPP LTE - Advanced Physical Layer, 2009. Retrieved January 21, 2015 from http://www.3gpp.org/ftp/workshop/2009-12-17_ITU-R_IMT-Adv_eval/docs/pdf/REV-090003-r1.pdf.

[2] Banks, J., and Carson, J. S. Introduction to discrete-event simulation. In *Proceedings of the 18th conference on Winter simulation - WSC '86* (New York, New York, USA, Dec. 1986), ACM Press, pp. 17–23. DOI: 10.1145/318242.318253.

[3] Berrou, C., Glavieux, a., and Thitimajshima, P. Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1. *Proceedings of ICC '93 - IEEE International Conference on Communications 2*, 1 (1993). DOI: 10.1109/ICC.1993.397441.

[4] Bontu, C., and Illidge, E. DRX Mechanism for Power Saving in LTE. *Communications Magazine, IEEE 47*, 6 (2009), 48–55. DOI: 10.1109/MCOM.2009.5116800.

[5] Cahn, C. Combined Digital Phase and Amplitude Modulation Communication Systems. *IRE Transactions on Communications Systems 8*, 3 (1960), 150–155. DOI: 10.1109/TCOM.1960.1097623.

[6] Chang, R. W. Synthesis of Band-Limited Orthogonal Signals for Multichannel Data Transmission. *Bell Syst. Tech. J. 45*, 10 (1966), 1775–1796.

[7] Dahl, O.-J., and Nygaard, K. SIMULA: an ALGOL-based simulation language. *Communications of the ACM 9*, 9 (Sept. 1966), 671–678. DOI: 10.1145/365813.365819.

[8] Ergen, M. IEEE 802.11 Tutorial, 2002. Retrieved January 21, 2015 from http://ayman.elsayed.free.fr/msc_student/wlan-tutorial.pdf.

[9] Ermedahl, A. Discrete Event Simulation in Erlang. Master's thesis, Uppsala University, 1995. DOI: 10.1.1.40.4956.

[10] Frenger, P., Parkvall, S., and Dahlman, E. Performance comparison of HARQ with Chase combining and incremental redundancy for HSDPA. *IEEE 54th Vehicular Technology Conference. VTC Fall 2001. Proceedings (Cat. No.01CH37211) 3* (2001), 1829–1833. DOI: 10.1109/VTC.2001.956516.

[11] García-alís, D., Stirling, I., and Stewart, B. Introduction to LTE 3GPP Evolution. Retrieved August 4, 2013 from `http://www.steepestascent.com/content/mediaassets/pdf/presentations/SA_Introduction_to_LTE.pdf`.

[12] Gast, M. S. *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2005.

[13] Henderson, T. R., and Riley, G. F. Network Simulations with the ns-3 Simulator. In *SIGCOMM* (2008).

[14] Holma, H., and Toskala, A. *LTE for UMTS: OFDMA and SC-FDMA based radio access*. John Wiley & Sons, Ltd., 2009.

[15] Hu, Z., Susitaival, R., Chen, Z., Fu, I. K., Dayal, P., and Baghel, S. Interference avoidance for in-device coexistence in 3GPP LTE-advanced: Challenges and solutions. *IEEE Communications Magazine 50*, 11 (Nov. 2012), 60–67. DOI: 10.1109/MCOM.2012.6353683.

[16] Jäntti, R. Wireless communications primer for automation engineers Part II : Medium access control, 2013. Retrieved January 21, 2015 from `https://noppa.aalto.fi/noppa/kurssi/as-74.3199/luennot/AS-74_3199_slides_9.pdf`.

[17] Jing, Z., Waltho, A., Xue, Y., and Xingang, G. Multi-radio coexistence: Challenges and opportunities. In *Proceedings - International Conference on Computer Communications and Networks, ICCCN* (Aug. 2007), IEEE, pp. 358–364. DOI: 10.1109/ICCCN.2007.4317845.

[18] Kiminki, S., and Hirvisalo, V. Coexistence-aware scheduling for LTE and WLAN during hard in-device interference. In *Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)* (2012), IEEE, pp. 1–6. DOI: 10.4108/icst.crowncom.2012.249458.

[19] Kiminki, S., and Hirvisalo, V. In-Device Coexistence Simulation for Smartphones. In *Proceedings of the 27th European Conference on Modelling and Simulation (ECMS 2013)* (2013), pp. 538–543.

[20] Kiminki, S., Piipponen, A.-V., Zetterman, T., Knuuttila, J., and Hirvisalo, V. Method, apparatus, and computer program product for coexistence-aware communication mechanism for multi-radios, Aug. 2013. Patent No. US20130225068.

[21] Law, A. M., and Kelton, W. D. *Simulation Modeling and Analysis*. McGraw-Hill, Nov. 1999.

[22] Metropolis, N., and Ulam, S. The Monte Carlo Method. *Journal of the American Statistical Association 44*, 247 (Apr. 1949).

[23] Myung, H. G., Lim, J., and Goodman, D. J. Peak-to-average power ratio of single carrier FDMA signals with pulse shaping. *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, PIMRC* (2006), 3–7. DOI: 10.1109/PIMRC.2006.254407.

[24] Myung, H. G., Lim, J., and Goodman, D. J. Single carrier FDMA for uplink wireless transmission. *IEEE Vehicular Technology Magazine 1* (2006), 30–38. DOI: 10.1109/MVT.2006.307304.

[25] Nyquist, H. Certain Topics in Telegraph Transmission Theory. *Transactions of the American Institute of Electrical Engineers 47*, 2 (Apr. 1928), 617–644. DOI: 10.1109/T-AIEE.1928.5055024.

[26] Oracle Corporation. The Java Programming Language. Retrieved January 21, 2015 from `http://java.com/en/`.

[27] Priyanto, B. E., Codina, H., Rene, S., Sø rensen, T. B., and Mogensen, P. Initial performance evaluation of DFT-spread OFDM based SC-FDMA for UTRA LTE uplink. In *IEEE Vehicular Technology Conference* (2007), IEEE, pp. 3175–3179. DOI: 10.1109/VETECS.2007.650.

[28] Razavi, B. *RF microelectronics*. Prentice Hall, Jan. 1997.

[29] Sargent, R. G. Verifying and validating simulation models. In *Winter Simulation Conference* (New York, New York, USA, Nov. 1996), ACM Press, pp. 55–64. DOI: 10.1145/256562.256572.

[30] Schwetman, H. CSIM. In *Proceedings of the 18th conference on Winter simulation - WSC '86* (New York, New York, USA, Dec. 1986), ACM Press, pp. 387–396. DOI: 10.1145/318242.318464.

[31] Shannon, R. Introduction to the art and science of simulation. *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274) 1* (1998), 7–14. DOI: 10.1109/WSC.1998.744892.

[32] Singh, S., and Nemani, C. Fluent Interfaces. *ACEEE Int. J. on Information Technology 01*, 2 (2011).

[33] The Apache Software Foundation. Apache Maven, 2014. Retrieved January 21, 2015 from `http://maven.apache.org`.

[34] Weinstein, S., and Ebert, P. Data Transmission by Frequency-Division Multiplexing Using the Discrete Fourier Transform. *IEEE Transactions on Communication Technology 19*, 5 (1971), 628–634. DOI: 10.1109/TCOM.1971.1090705.

[35] Zetterman, T., Piipponen, A.-V., Kiminki, S., Knuuttila, J., and Hirvisalo, V. Methods and Apparatus for In-Device Coexistence, Aug. 2013. Patent No. US20130194985.